

UNIVERSIDAD DE BUENOS AIRES

FACULTAD DE CIENCIAS EXACTAS Y NATURALES

LICENCIATURA EN CIENCIAS DE LA COMPUTACIÓN

---

# Teóricas de Ingeniería del Software II

---

*Autor:*

Julián SACKMANN

10 de Septiembre de 2012



**Facultad de Ciencias Exactas y  
Naturales**

**Universidad de Buenos Aires**

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

<http://exactas.uba.ar>

# Índice

<b>1</b>	<b>Introducción: ¿Qué es la Ingeniería del Software?</b>	<b>5</b>
	Dijkstra . . . . .	5
	Yourdon . . . . .	5
1.1	Ingeniería . . . . .	5
1.1.1	Evolución de una Ingeniería . . . . .	5
1.2	Ingeniería... de software . . . . .	6
1.2.1	Definición . . . . .	6
1.2.2	Negación de la ingeniería . . . . .	7
1.2.3	Ciencias de la computación vs Ingeniería del software . . . . .	7
1.2.4	Church vs Turing . . . . .	7
<b>2</b>	<b>Modelos de ciclo de vida</b>	<b>8</b>
2.1	Code & Fix . . . . .	8
2.2	Modelo en cascada . . . . .	9
2.2.1	Problemas . . . . .	9
2.2.2	Mejoras . . . . .	10
2.2.3	Diseño preliminar . . . . .	10
	Sashimi . . . . .	10
	Subproyectos . . . . .	11
	Prototipos . . . . .	12
<b>3</b>	<b>Modelo iterativo incremental</b>	<b>12</b>
3.1	Ventajas y desventajas . . . . .	13
3.2	Variaciones . . . . .	14
3.3	Métodos . . . . .	14
3.3.1	Modelo en Espiral . . . . .	14
3.3.2	RUP . . . . .	15
3.3.3	SCRUM (Introducción) . . . . .	15
	Co-evolución . . . . .	15
<b>4</b>	<b>Métodos Ágiles</b>	<b>16</b>
4.0.4	Agile Manifesto . . . . .	16
4.0.5	Conceptos . . . . .	16
	Time Boxing . . . . .	17
	Desarrollo Incremental . . . . .	17
	Desarrollo Iterativo . . . . .	17
4.0.6	Principios . . . . .	17
4.0.7	Diferencias principales con métodos tradicionales . . . . .	18
<b>5</b>	<b>Scrum</b>	<b>18</b>
5.1	¿En qué proyectos conviene utilizarlo? . . . . .	18
5.1.1	Costo de cambio . . . . .	19
5.2	Roles . . . . .	19
	Product Owner . . . . .	19
	Scrum Master . . . . .	20
	Equipo . . . . .	20
5.3	Diagrama general . . . . .	20
5.4	Producto y Product backlog . . . . .	20
	Ejemplo de product backlog . . . . .	21
5.5	User Stories . . . . .	22

5.5.1	INVEST . . . . .	22
5.5.2	Ejemplo de user story . . . . .	22
5.6	Inicio de un proyecto . . . . .	23
5.7	Sprints . . . . .	23
5.8	Herramientas . . . . .	23
5.8.1	Pizarra . . . . .	23
5.8.2	Mediciones . . . . .	24
	Product burndown chart . . . . .	24
	Story burndown chart . . . . .	24
5.9	Problemas de scrum . . . . .	24
<b>6</b>	<b>Gestión de proyectos</b>	<b>25</b>
6.1	¿A qué se dedica un gerente? . . . . .	25
6.1.1	Identificación de Stakeholders . . . . .	25
6.1.2	Determinación de factores críticos . . . . .	26
6.1.3	Identificación preliminar de requerimientos . . . . .	26
6.1.4	Otros . . . . .	26
6.2	WBS . . . . .	27
6.2.1	Tipos de WBS . . . . .	27
	WBS de proceso . . . . .	27
	WBS de producto . . . . .	27
	WBS híbrido . . . . .	27
6.2.2	Dependencias . . . . .	28
	Tareas especiales . . . . .	28
	Dependencias externas . . . . .	28
6.2.3	Camino crítico . . . . .	28
6.2.4	Diagrama de Gantt . . . . .	29
6.2.5	Otras definiciones . . . . .	30
6.3	Gestión de riesgos . . . . .	30
6.3.1	Sistematización . . . . .	30
6.3.2	Identificación de riesgos . . . . .	31
6.3.3	Documentación . . . . .	31
6.3.4	Planificación . . . . .	32
6.4	Plan de gestión de proyecto . . . . .	32
<b>7</b>	<b>Atributos de calidad</b>	<b>34</b>
7.1	Generales vs Concretos . . . . .	34
7.2	Los 6 atributos . . . . .	35
7.2.1	Disponibilidad . . . . .	35
	Tipos de fallas . . . . .	35
	Medidas para hacer un sistema más tolerante a fallas . . . . .	35
	Ejemplo . . . . .	35
7.2.2	Modificabilidad . . . . .	36
	Ejemplo . . . . .	36
7.2.3	Performance . . . . .	36
	Ejemplo . . . . .	37
7.2.4	Seguridad . . . . .	37
	Ejemplo . . . . .	37
7.2.5	Usabilidad . . . . .	37
	Testeabilidad . . . . .	38
7.2.6	Testeabilidad . . . . .	38
7.3	Especificación . . . . .	38

7.3.1	Ejemplos . . . . .	38
7.3.2	Disponibilidad . . . . .	38
7.3.3	Modificabilidad . . . . .	39
7.3.4	Performance . . . . .	39
7.3.5	Seguridad . . . . .	39
7.3.6	Usabilidad . . . . .	40
7.4	Quality Attribute Workshops . . . . .	40
<b>8</b>	<b>UML</b>	<b>41</b>
<b>9</b>	<b>Unified Process</b>	<b>41</b>
9.0.1	Buenas prácticas . . . . .	41
9.1	Iteraciones . . . . .	41
9.1.1	Fases . . . . .	41
9.1.2	Hitos . . . . .	42
9.1.3	Fase de inepción . . . . .	42
9.1.4	Fase de elaboración . . . . .	42
9.1.5	Fase de construcción . . . . .	43
9.1.6	Fase de transición . . . . .	43
9.2	Ventajas y desventajas . . . . .	43
<b>10</b>	<b>Arquitecturas</b>	<b>43</b>
10.1	Definiciones . . . . .	44
10.2	Los tres principios fundamentales . . . . .	44
10.3	¿Qué hace que una arquitectura sea buena? . . . . .	44
10.4	Documentación de una arquitectura . . . . .	45
10.5	Arquitectura vs diseño . . . . .	45
10.6	ADD . . . . .	45
<b>11</b>	<b>Estilos arquitectónicos</b>	<b>46</b>
11.1	Ventajas . . . . .	46
11.2	Taxonomía . . . . .	46
11.3	Estilos . . . . .	46
11.3.1	Data Flow . . . . .	46
	Batch sequential . . . . .	47
	Pipe and filter . . . . .	47
	Ventajas y desventajas . . . . .	47
11.3.2	Call return . . . . .	47
11.3.3	Layered . . . . .	48
11.3.4	Client Server . . . . .	48
11.3.5	Componentes independientes . . . . .	48
	Invocación implícita vs explícita . . . . .	48
11.3.6	Centradas en datos . . . . .	49
11.3.7	Basadas En Eventos . . . . .	49
11.4	Viewtypes . . . . .	49
11.4.1	De módulos . . . . .	50
	Utilidad . . . . .	50
	Cuando no . . . . .	51
	Ejemplo . . . . .	51
11.4.2	De componentes y conectores . . . . .	51
	Ejemplo . . . . .	52
11.4.3	De estructuras de asignación . . . . .	52

---

<b>12 Tácticas</b>	<b>52</b>
12.1 Relación con los estilos . . . . .	52
12.2 Tácticas para los atributos . . . . .	52
12.2.1 Disponibilidad . . . . .	52
12.2.2 Performance . . . . .	53
12.2.3 Seguridad . . . . .	53
12.2.4 Modificabilidad . . . . .	54
12.2.5 Usabilidad . . . . .	54
Reglas de oro de la usabilidad . . . . .	54
<b>13 Fuentes</b>	<b>56</b>

# 1 Introducción: ¿Qué es la Ingeniería del Software?

Existen dos corrientes contrapuestas de lo que es la ingeniería del software:

## Dijkstra

Una, que tiene como cabeza a Dijkstra afirma que la ingeniería del software es “aquello que hacemos porque no sabemos programar bien”. Dijkstra llama a la ingeniería del software “*the doomed discipline*”: está perdida porque sus objetivos son contradictorios. Dijkstra plantea que el enfoque correcto a la programación (y particularmente a su enseñanza) es el de ver a los programas como fórmulas matemáticas y el trabajo del programador es derivar esa fórmula mediante la manipulación de símbolos. Tiene que ser un curso en matemática formal. Incluso opina que debería cambiarse el nombre a VLSAL: “Very Large Scale Application of Logic”.

La principal crítica que se le puede realizar a esta visión es que es demasiado academicista. No siempre es aplicable la visión de un programa como una fórmula matemática (menos aún cuando se trata de un proyecto grande). Del mismo modo, si bien no siempre es factible **demostrar** la correctitud de un programa, tiene valor intentar hacerlo (incluso escribirlos de forma de que sea posible su demostración).

## Yourdon

La otra visión, encabezada por Yourdon, en la que se plantea una dicotomía fuerte entre la ciencia y la ingeniería. Según Yourdon, la construcción de software dista mucho de las fórmulas matemáticas y las demostraciones que plantea Dijkstra, sino que son “un conjunto de métodos prácticos que traten con la naturaleza propesa a errores del personal de proyecto, los encargados de mantenimiento, los usuarios, etc”.

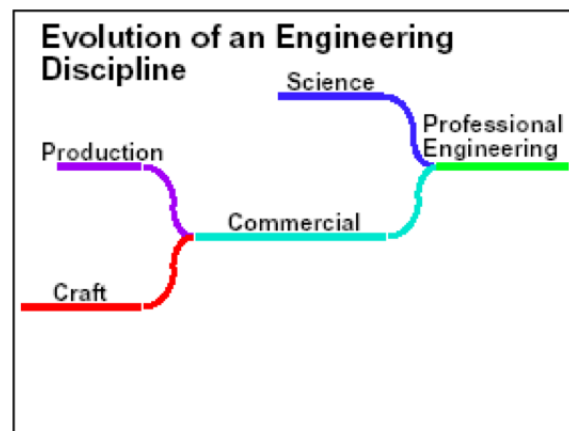
Esta visión fue llevada al extremo por Fred Brooks, que plantea si realmente el trabajo de un computador es el de *hacer ciencia*: tanto un ingeniero como un científico construyen, pero su propósito esencial es distinto: el ingeniero *aprende para construir*, mientras que el científico *construye para aprender*.

Brooks opina que cuando una profesión comienza, es habitual confundir las herramientas que uno usa con hacer ciencia y que “computer science” tiene poco que ver con la computadora. Siguiendo esta misma línea, Zemanek opina que la computación es la “ingeniería de los objetos abstractos”.

## 1.1 Ingeniería

La definición de “ingeniería” (no acotada la software) es crear soluciones eficientes a problemas prácticos, aplicando conocimiento científico construyendo cosas al servicio de la humanidad. “Es ciencia con un propósito”. Permite a la gente común hacer cosas que antes requerían virtuosos.

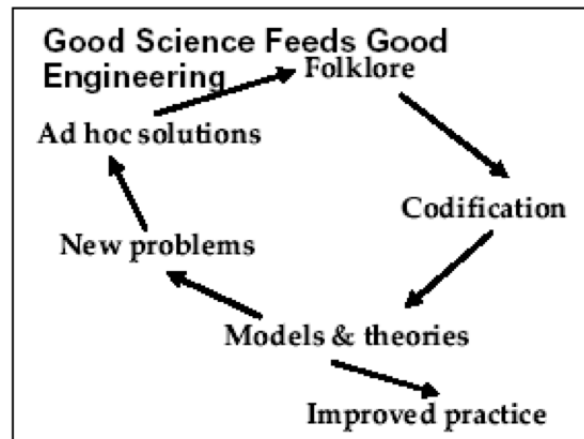
### 1.1.1 Evolución de una Ingeniería



Según Shaw, toda ingeniería pasa por una serie de etapas hasta convertirse en tal:

- Comienza siendo un ***craft*** (“habilidad”). En esta etapa se construyen cosas en base a intuición y prueba y error. No existen métodos formales para la construcción. Se construyen cosas con el objeto de **usarlas**.
- Cuando se inyecta la veta de producción a esto, se transforma en algo **comercial**. Se comienza a tener más habilidad y regularidad en la construcción (procesos más establecidos), pero aún sin formalismos. Se construyen cosas con el objeto de **vender**.
- A medida que esto avanza, la ciencia se involucra por los métodos de construcción y se forma la **ingeniería profesional**, con profesionales educados en la ciencia, formalismos, análisis, teorías, etc. Se segmenta el conocimiento.

La aparición de modelos y teorías es clave para alimentar el ciclo que nos llevará a la verdadera disciplina de ingeniería: es un circuito retroalimentado:



Cuando uno está construyendo sistemas, aparecen problemas nuevos (ejemplo actual: *Big Data*) a las que se van encontrando soluciones ad-hoc. Eso genera un cierto “folklore” sobre esas soluciones ad-hoc hasta que en algún momento se hacen modelos y teorías para solucionarlo. Pero esto a su vez hace aparecer nuevos problemas, con lo que se crean nuevas soluciones...

## 1.2 Ingeniería... de software

### 1.2.1 Definición

Según la **IEEE**, la ingeniería del software es “la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, mantenimiento y operación de sistemas de software”. Coloquialmente, hace referencia a la **aplicación de prácticas ingenieriles para la construcción de software**.

La ingeniería trata con:

- Requerimientos.
- Diseño (soluciones técnicas).
- Construirlo.
- Testearlo.
- Mantenimiento.

- Gestión
  - de configuración.
  - de proyecto.
- Procesos.

Un problema de la ingeniería del software es que el desarrollo no existe en el éter, sino en un marco de *deadlines* temporales, económicos.

### 1.2.2 Negación de la ingeniería

Existe gente que cree que la ingeniería del software en realidad “no existe”. Sus argumentos suelen venir de tres vertientes:

- Falta formalidad en la “ciencia” que está por detrás. Todavía está demasiado inmadura. Esta ingeniería podría existir alguna vez, pero por ahora es solamente un conjunto de prácticas.
- El enfoque “ingenieril” está errado desde el vamos: escribir software es más similar a un arte que a una ingeniería.
- El software es algo abstracto y la ingeniería se ocupa de cosas concretas.

### 1.2.3 Ciencias de la computación vs Ingeniería del software

Cuando hablamos de **ciencias de la computación** nos referimos a las **teorías y los fundamentos**.

Cuando hablamos de **ingeniería de software**, hablamos de **aspectos prácticos de desarrollar y entregar software útil**. Observemos que la ingeniería del software se nutre de la ciencia de la computación.

Observemos que la ingeniería en sistemas ( $\neq$  “de software”) trata sobre la construcción de sistemas que combinan hardware y software.

### 1.2.4 Church vs Turing

- Turing tiene una idea más mecanicista, en la que para programar se toman decisiones de diseño basados en los problemas intrínsecos de la máquina donde corren. Ej: `C/C++`.
- Church tiene una idea más algebraica, basada en lambda cálculo, en la que se intenta obtener un análisis de forma más “pura” del problema de la realidad. Ej: `Lisp`.

La ingeniería del software:

- Trabaja con algo muy complejo.
- Que trata con una problemática muy amplia.
- Que tiene dificultades esenciales.
- Está evolucionando.

## 2 Modelos de ciclo de vida

Un modelo de desarrollo de software se define como **un marco de trabajo usado para estructurar, planear y controlar el proceso de desarrollo de un sistema de información**. Sirve para **planificar, organizar y ejecutar** un proyecto.

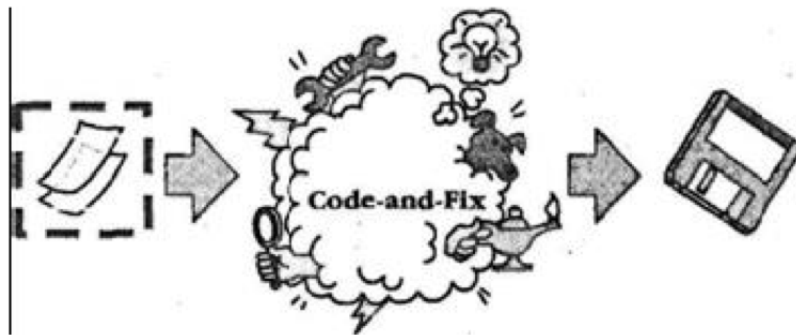
Los modelos de ciclo de vida tratan el tema de cómo organizar un proceso de desarrollo de software en cuanto a su estructura:

- Etapas de un proyecto.
- Orden relativo.
- Criterios de transición entre etapas.

Cambiando entre distintos modelos de ciclos de vida se obtienen *tradeoffs* entre:

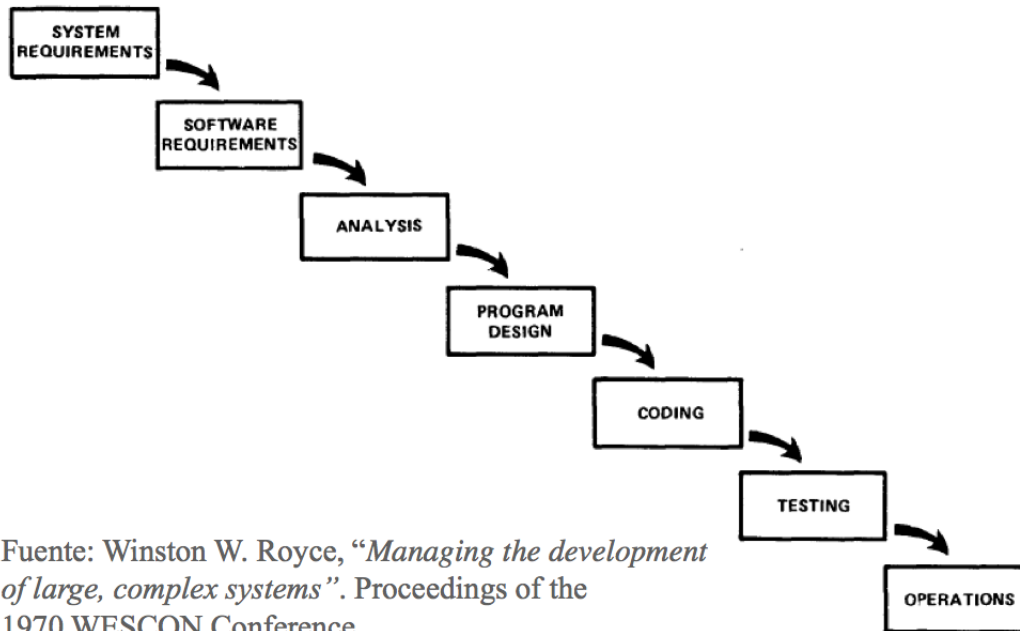
- Velocidad de desarrollo.
- Calidad del producto.
- Visibilidad del proyecto.
- Posibilidad de implementar versiones intermedias.
- Carga de trabajo administrativo y de gestión.
- Documentación (tipo y cantidad).
- Exposición al riesgo.
- Relación con el cliente.

### 2.1 Code & Fix



Es la ausencia de ciclo de vida. Se suele utilizar bastante en proyectos de desarrollo chicos. Programación, arregla, empatcha. El principal problema de este método es que no escala bien: con un proyecto de tamaño grande ya se vuelve caótico e inusable.

## 2.2 Modelo en cascada



En su versión clásica, postulada por Royce, el proceso de desarrollo de software consta de etapas:

- Analizar los requerimientos del sistema.
- Extraer los requerimientos.
- Analizarlos.
- Diseñar.
- Codificar.
- Testear.
- Desplegar.

Una de las premisas más fuertes que tiene este modelo es que uno ya sabe lo que está haciendo con el software. El programador ya es experto del dominio.

El criterio de transición entre etapas del modelo en cascada es **documental**: sólo se pasa a la siguiente etapa cuando toda la documentación referida a la etapa actual está terminada.

### 2.2.1 Problemas

Este modelo trae aparejados varios problemas:

- Idealista pensar en identificar correctamente todos los requerimientos al principio.
- No permite implementaciones parciales.
- Usuario sólo involucrado al principio y al final.
- Se retrasa la detección de problemas críticos.
- En el caso de encontrarlos, es difícil volver para atrás para arreglarlos.

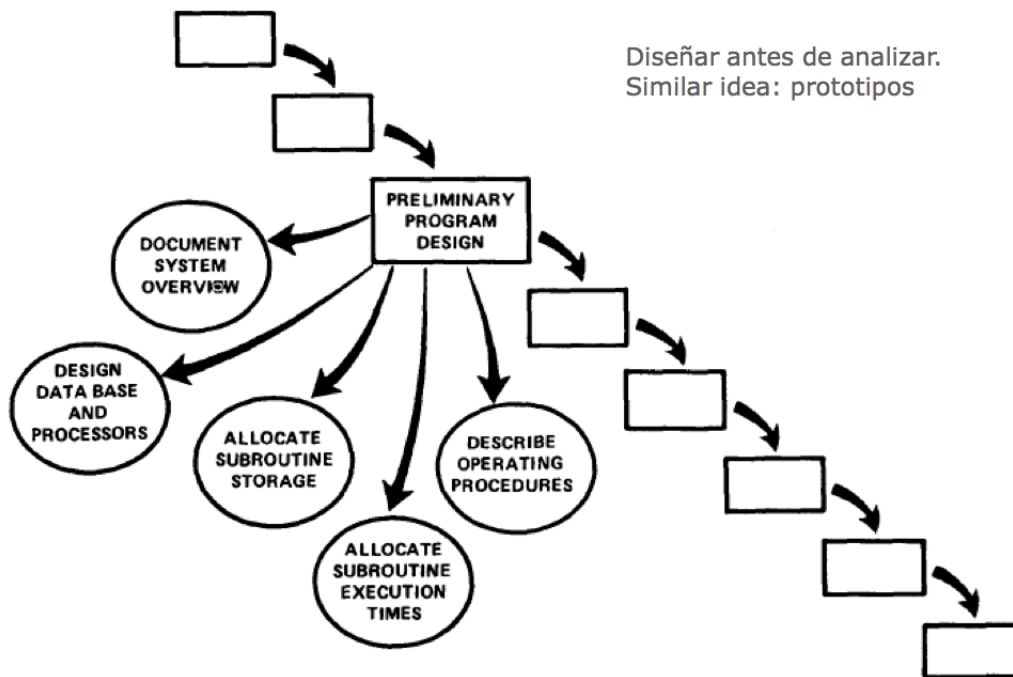
- Lo que se aprende en una etapa no se vuelve a revisar.

Jackson critica el primer punto de esto haciendo una analogía con el principio de incertidumbre de Heisenberg: “cualquier actividad de desarrollo de software inevitablemente cambia el entorno del cual surgió la necesidad del software”.

En la práctica, de cada etapa siempre se termina volviendo (con dificultad) a una anterior.

### 2.2.2 Mejoras

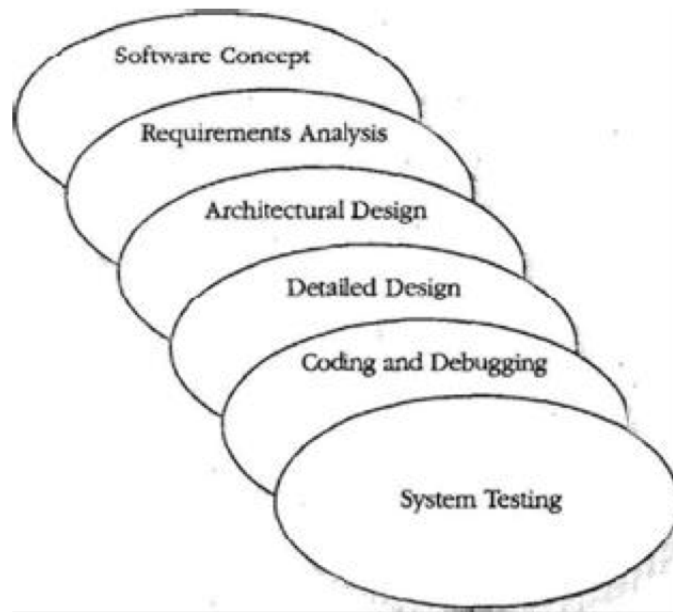
### 2.2.3 Diseño preliminar



Una posible mejora consiste en hacer un análisis preliminar antes de comenzar la etapa de análisis. De esta forma se pueden detectar problemas que pueden surgir en el diseño en forma temprana e impactan el análisis.

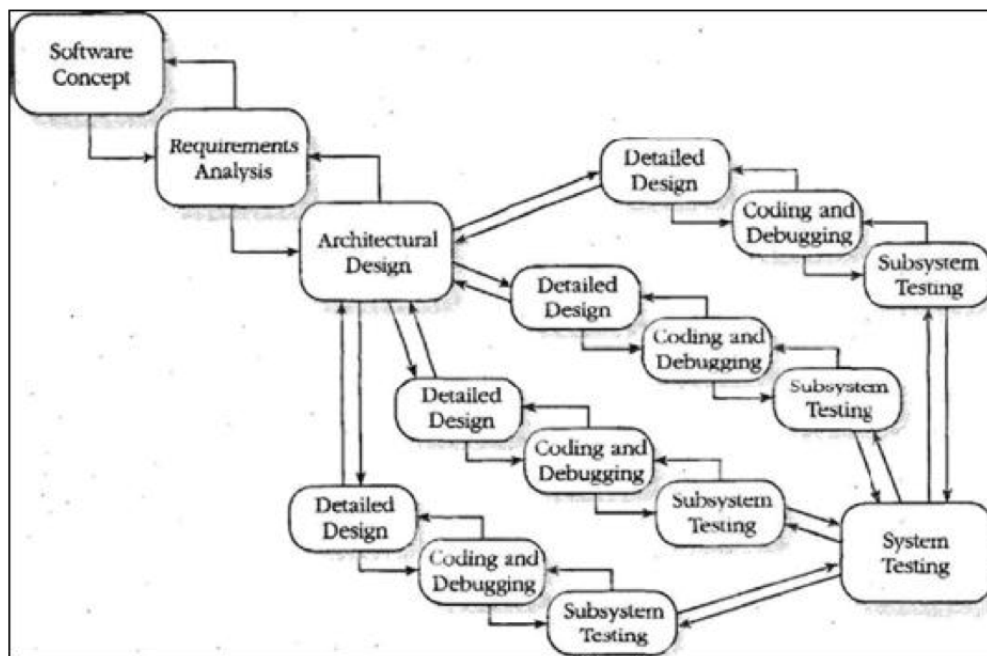
Esta técnica si bien presenta una mejora ante el modelo en cascada clásico, es más un parche que un cambio de base.

### Sashimi



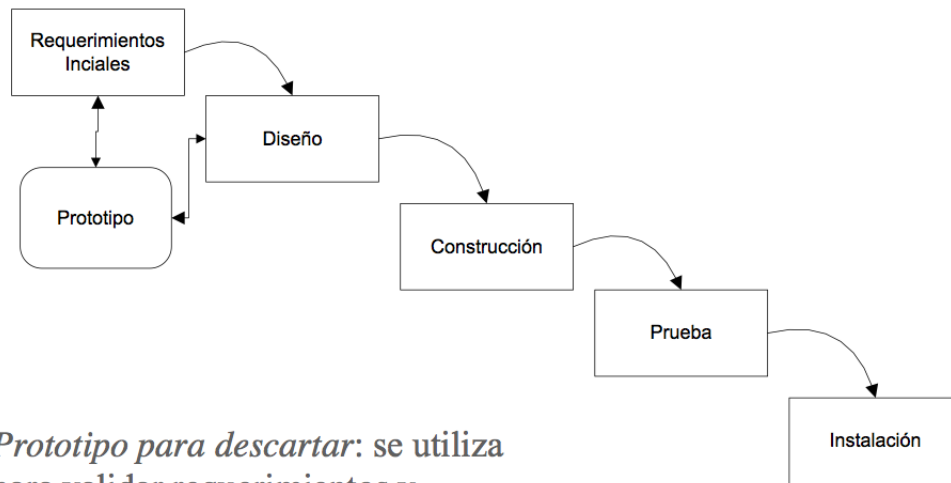
El modelo en cascada **sashimi** (o “de fases superpuestas”) intenta atacar el problema de la dificultad intrínseca del modelo en cascada clásico a la hora de volver a una etapa anterior. Para eso, flexibiliza el criterio y permite que se superpongan parcialmente dos etapas consecutivas. De este modo, se simplifica el proceso de volver a la anterior en el caso de encontrarse un problema.

### Subproyectos



El modelo en cascada **con subproyectos** plantea seguir el modelo en cascada clásico hasta la parte de diseño general. De ahí en adelante se continúa con varias alternativas durante las etapas de **diseño detallado**, **codificación** y **testeo**. Concluidas esas etapas se comparan los resultados obtenidos y se continúa, unificadamente con una etapa de testeo global del sistema.

## Prototipos



*Prototipo para descartar: se utiliza para validar requerimientos y resolver aspectos críticos del diseño*

La idea del modelo de **prototipos** (o “de reducción de riesgo”) es hacer un diseño preliminar antes de empezar el análisis real: cuando uno identifica los requerimientos iniciales, presenta al usuario con un prototipo de lo “visible” para el usuario y/o partes críticas. De este modo se obtiene una validación de parte del cliente de la correctitud de la idea.

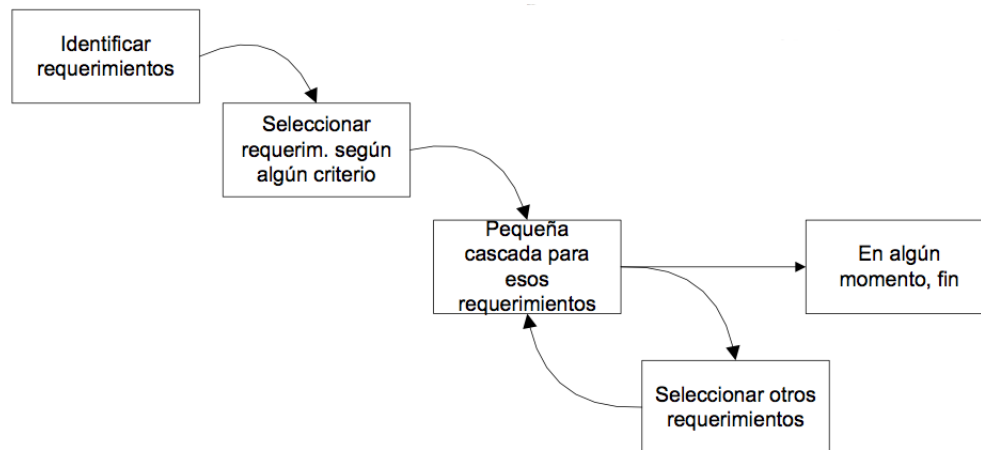
Esto, en proyectos chicos funciona bien, pero no escala.

ewline

## 3 Modelo iterativo incremental

El Modelo **iterativo incremental** (también llamado “evolutivo”) es un proceso de desarrollo de software, creado en respuesta a las debilidades del modelo tradicional de cascada. Su nombre se origina de:

- **Iterativo:** se hace varias veces lo mismo.
- **Incremental:** el producto “aumenta” a medida que avanzamos.



Los modelos iterativos incrementales constan de las siguientes etapas:

- **Identificación de requerimientos iniciales** (el nivel de detalle de esto depende del **método**, no del **modelo**).
- **Selección de un subconjunto de requerimientos**: el criterio depende del método. Algunos posibles son:
  - Lo más visible para el cliente.
  - Lo que más rápido determine viabilidad del proyecto.
  - Lo más crítico.
  - Lo más rápido de terminar.
- **Desarrollo de los requerimientos elegidos**: se puede realizar mediante un pequeño método en cascada, mediante TDD (*test driven development*) u otras técnicas.
- **Seleccionar otro subconjunto de requerimientos e iterar**.

### 3.1 Ventajas y desventajas

#### Ventajas:

- El usuario ve resultados rápidamente.
- *Feedback* rápido.
- Se piensa en la calidad del sistema desde el inicio.
- Se pueden atacar más fácilmente los riesgos.
- Los ciclos van mejorando con las experiencias de los anteriores.
- Minimizan la cantidad de funciones que se desarrollan y no se usan.

#### Problemas:

- Requiere un cliente involucrado durante todo el proceso de desarrollo.
- Se asume que todos los programadores tienen alto *seniority*.
- No son prácticos en proyectos donde los requerimientos están muy establecidos desde el principio.

## 3.2 Variaciones

Un modelo iterativo incremental requiere ser instanciado con un método. Los distintos métodos que siguen el modelo iterativo incremental presentan diferencias en numerosas características. Por ejemplo:

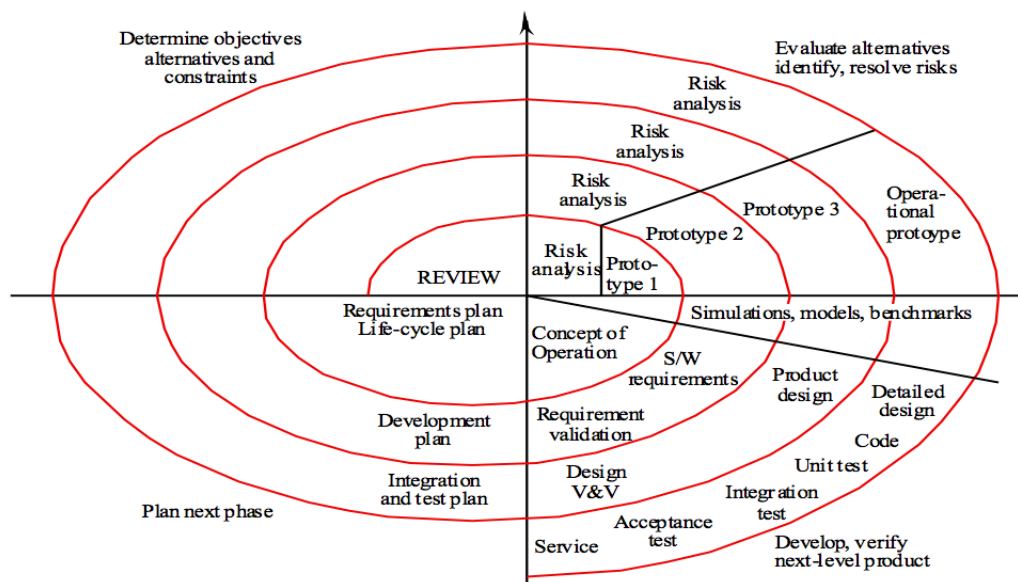
- Duración de las iteraciones.
- Transición de etapas.
- Existencia o no de **tipos** de iteraciones (iteraciones más orientadas a una actividad que a otra).
- Cantidad de esfuerzo dedicada a la elicitación inicial de requerimientos.
- Actitud defensiva ante cambios en los requerimientos.

## 3.3 Métodos

Un **método de desarrollo** es una descripción de qué, como y cuándo hay que hacer algo. Que perfiles deben hacerlo, cómo se hace el testing, cómo se documenta, etc.

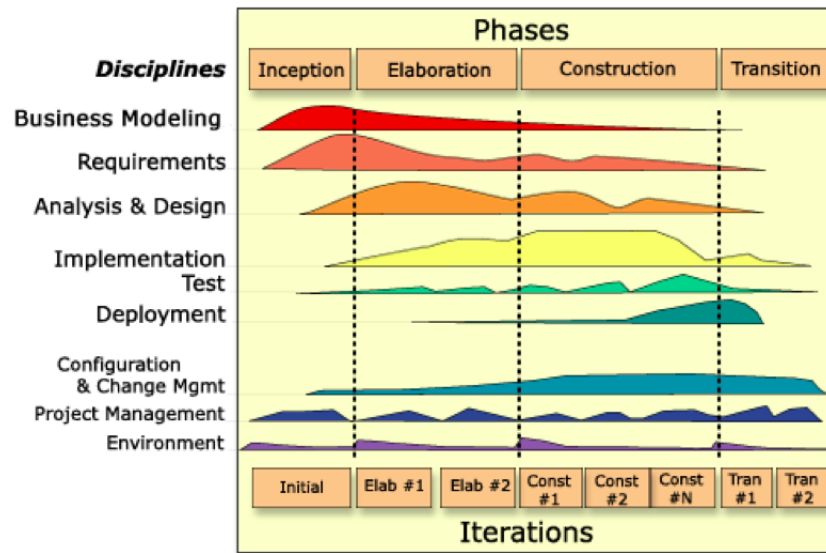
### 3.3.1 Modelo en Espiral

Propuesto por Boehm. El gráfico atrae mucho, pero nunca fue demasiado popular. El riesgo es el que guía la elección de funcionalidad.



Se concibe el desarrollo de software como una espiral infinita y creciente en donde cíclicamente se alternan períodos de **análisis, evaluación, desarrollo y planeamiento**.

### 3.3.2 RUP



Su nombre viene de “*Rational Unified Proces*”.

Plantea que el software se desarrolla en iteraciones, luego de cada una de las cuales tengo un pedazo del sistema, cada vez más grande. Las iteraciones son distintas (y tienen distinto nombre): primero hay una etapa de iteración inicial donde uno no entiende los requerimientos ni el dominio. Entonces la mayor parte del tiempo va a ser entender esas cosas.

Cuanto más avanzo en las iteraciones más conocimiento tengo del dominio y más voy a dedicar a diseñar. Luego hay iteraciones de diseño, construcción, transición, etc.

### 3.3.3 SCRUM (Introducción)

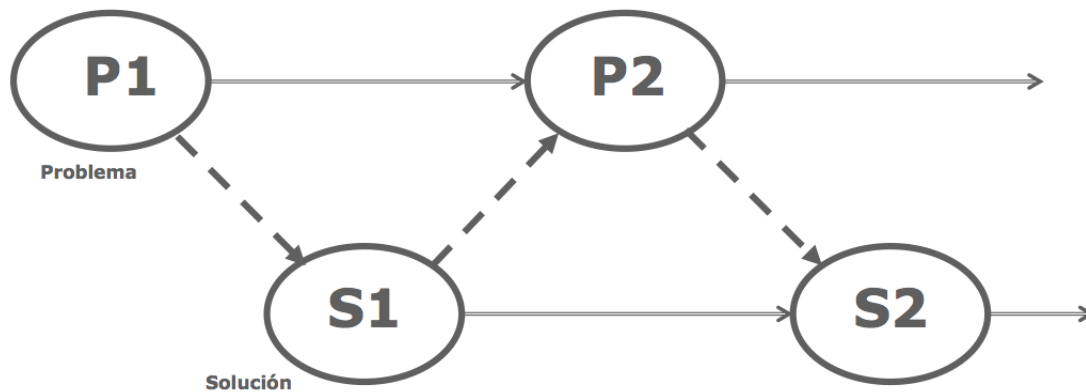
Es un método ágil que usa el ciclo de vida iterativo incremental. Propone definir una lista de requerimientos (**backlog**), seleccionar algunos, descomponerlo en tareas (de 1 a 4 semanas) y realizar iteraciones con chequeos diarios generando incremento de producto.

En teoría, no hay distinción de tipo de iteraciones, son todas iguales. En la práctica sin embargo, suele haber iteraciones más enfocadas a investigación, a implementación, a testeo, etc. Sin embargo, en todas las iteraciones se hace “un poco de todo”. Por ejemplo:

- Inestabilidad de requerimientos / novedad del producto / Innovación ⇒ Mayor peso al enfoque evolutivo.
- Posibilidad concreta de partir el desarrollo ⇒ Iteraciones más cortas.
- Arquitectura más compleja o tecnología no conocida ⇒ Enfoque de atacar riesgos desde el inicio, cuidado con la “self emergence”.
- Mayor complejidad del negocio ⇒ No descuidar la especificación.

### Co-evolución

El concepto de **co-evolución** fue presentado por Fred Brooks. Posulta que *el espacio del problema evoluciona a medida que el espacio de la solución evoluciona al ser explorado*.



Los ciclos de vida deben ser co-evolutivos: arrancar con un problema y se le encuentra una solución. Con el problema original y esta solución, sale un nuevo problema. El espacio del problema evoluciona a medida que el espacio de las soluciones evoluciona al ser explorado.

## 4 Métodos Ágiles

Los métodos ágiles son un conjunto de métodos que implementan modelo iterativo incremental.

### 4.0.4 Agile Manifesto

El **ágile manifesto** es un documento que surge como reacción de la comunidad de desarrollo ante la crisis del software. En ese momento se creía que era un mero problema organizativo y de gestión. El remedio que se quiso aplicar es armar un marco de gestión muy importante, agregando mucha formalidad y abundancia de documentación al proceso de desarrollo. Esto se dio mediados y fines de los 80 y casi todos los 90.

El problema es que hubo una sobrecarga de esta burocracia, lo que impulsó un movimiento para el otro lado. Cambiar la tendencia de desarrollo de software proponiendo métodos de desarrollo que valoren más cosas como:

- **Individuos e interacciones sobre procesos y herramientas:** se decía de no confiar en un proceso de desarrollo con demasiadas herramientas y procesos. Confiemos más en la gente y dejémosla interactuar.
- **Software que funciona sobre documentación exhaustiva:** en esa época todos los procesos de desarrollo tenían mucha documentación. Esta gente propuso dejar de hacer tanta documentación, lo único imprescindible para tener un sistema andando es el software, no la documentación.
- **Colaboración con el cliente sobre negociación de contratos:** cuando uno hace un contrato llave en mano, por los cambios a requerimientos y cosas que uno no pudo prever la cosa nunca termina siendo como se querían. Esto generaba una serie de tensiones. Todos esos procesos del estilo de “esto es lo que hay que hacer y cualquier otra cosa es un desvío” son lo que se intenta cambiar.
- **Responder ante el cambio sobre seguimiento de un plan:** esto viene de la mano de lo anterior. En computación los planes son algo que cambia demasiado a diferencia de otras áreas.

Todos los métodos ágiles siguen estos lineamientos, que son bastante básicos. Proveen un marco básico desde donde empezar a trabajar y confían mucho en la gente para que se avance.

### 4.0.5 Conceptos

Los métodos ágiles introdujeron varios conceptos:

### Time Boxing

El *time boxing* consiste en tratar de ajustar las cosas en un período de tiempo acotado. Esto se plantea en contraposición al *feature boxing* de la planificación tradicional, en la que se hace un plan concreto para cumplir un objetivo.

Informalmente, *time boxing* se pregunta: “tengo este período de tiempo y estos recursos, ambos acotados. ¿Qué puedo encajar acá?”. Se intenta priorizar la duración sobre el alcance. La limitación estricta de tiempo estimula a mantener el foco. Esto hace que se genere un ritmo de desarrollo constante al que la gente se acostumbra. Se aplica esto a todo: iteraciones, reuniones, tareas grupales, individuales, etc.

### Desarrollo Incremental

El sistema va creciendo como consecuencia de la integración de nuevas funcionalidades. Cada funcionalidad que se agrega está testeada en forma unitaria, y también se prueba integrada al resto de la aplicación.

### Desarrollo Iterativo

El proyecto se divide en **iteraciones**, que el equipo encara como mini-proyectos en si mismos. Idealmente, al final de cada una de estas iteraciones se debe obtener como resultado un producto andando con una porción de la funcionalidad requerida ya diseñada, implementada, testeada, etc. Debería estar listo para usarse si ese fuera el último requerimiento.

En la práctica, esto no se suele aplicar en forma estricta, pero se respeta la idea general de que al finalizar una iteración no debería haber “pedazos” de código hecho “a medias”. Esto se diferencia muy notablemente del paradigma tradicional, en donde la integración era el último paso en el desarrollo de un sistema: acá se va integrando

#### 4.0.6 Principios

Los métodos ágiles introdujeron varios principios:

- **DRY: *Don't Repeat Yourself*.** No crear código, herramientas o infraestructura duplicadas.
- **KISS: *Keep It Simple Stupid!*.** Evitar la complejidad no esencial.
- **Do The Simplest Thing That Could Possibly Work.** Evitar la anticipación injustificada y las generalizaciones prematuras. Hay que mantener un balance entre generalizaciones que útiles y productivas y generalizaciones innecesarias y sin sentido.
- **YAGNI: *You Ain't Gonna Need It*.** Hace referencia a funcionalidades que el usuario cree que va a necesitar, pero realmente no lo va a necesitar.
- **DOGBITE: *Do it, Or it's Gonna Bite you In The End*.** Algunas cosas hay que hacerlas con anticipación (y suelen ser las menos atractivas).
- **SOC: *Separation of Concerns*.** Evitar el solapamiento de funcionalidad entre las diversas características o módulos de un programa.
- **Done-Done-Done:** ser sincero sobre el estado de terminación de una tarea o unidad de entrega.
- **Refactoring:** permanentemente reestructurar el código, el diseño, el proceso de desarrollo, etc.
- **Shipping is a feature, and your product must have it:** el producto de una iteración debería ser instalable por el cliente.

..y algunos anti-principios (un patrón organizacional de implementación o diseño que es popular pero contraproducente):

- **BDUF: *Big Design Up Front*.** Tiende a oponerse a YAGNI y KISS.

- **Optimización prematura** sin tener métricas que la justifiquen.
- **Bug Driven Development**: los requerimientos se completan en forma de bugs.

#### 4.0.7 Diferencias principales con métodos tradicionales

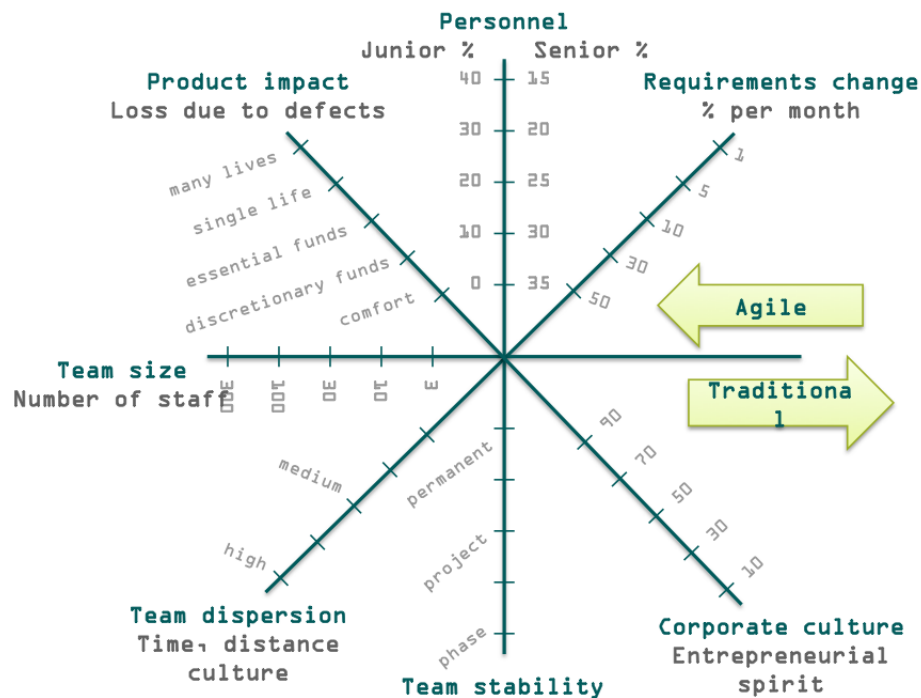
- **Tamaño**: los métodos ágiles conviene aplicarlos con equipos chicos, entre 5 y 9 integrantes. Para equipos más grandes, conviene apegarse a métodos tradicionales.
- **Ubicación**: los métodos ágiles requieren mucha comunicación y un equipo muy cohesivo. Si el equipo está disperso por muchos lugares geográficamente diversos, conviene quedarse con un método.
- **Estabilidad**: los métodos tradicionales son mejores para equipos altamente fluctuantes. Los métodos ágiles requieren un equipo de gente estable.
- **Seniority del equipo**: si el equipo es muy variado en cuanto a su nivel de seniority, conviene usar métodos tradicionales.
- **Cultura corporativa**: los métodos ágiles son mejores en contextos de alto espíritu emprendedor (ej. programar un juego para iPhone).
- **Cambios a requerimientos**: si los requerimientos son altamente cambiantes, conviene utilizar métodos ágiles, con iteraciones cortas.
- **Nivel de criticidad del software**: hace referencia a las pérdidas por bug. Para software de alto nivel de criticidad, conviene usar métodos más tradicionales porque son más estables y seguros.

## 5 Scrum

Como se mencionó en la sección 3.3.3, **Scrum** es un método ágil que implementa el ciclo de vida iterativo incremental para el desarrollo de software.

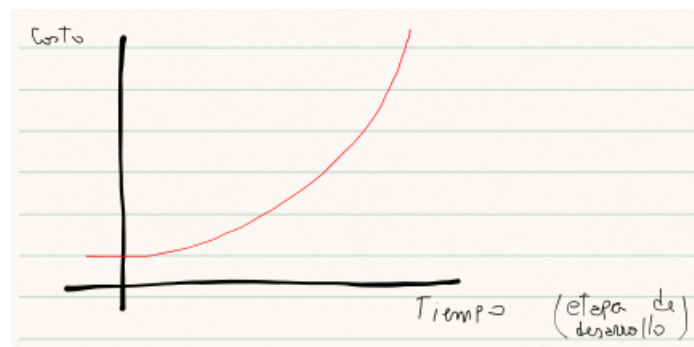
### 5.1 ¿En qué proyectos conviene utilizarlo?

El siguiente diagrama presenta en forma gráfica un lineamiento de cuándo conviene usar scrum y cuándo conviene usar métodos tradicionales, en función de algunas variables del proyecto. Cuánto más cerca se está del centro más conviene usar scrum.



### 5.1.1 Costo de cambio

El costo de cambiar cosas en el código sigue una función exponencial en el tiempo:



Hoy en día esta curva es un poco más aplanada porque los avances tecnológicos simplificaron mucho el cambiar código en producción. Sin embargo en esencia sigue siendo cierta.

## 5.2 Roles

Una de las principales características de scrum es la presencia de **roles** para la organización del personal. Scrum define 3 roles:

### Product Owner

El **product owner** es un rol individual que representa al cliente o usuario final del producto. Su principal función es la de definir las prioridades de negocio. Es el encargado de escribir los documentos con los que se guiará el equipo para desarrollar el proyecto. En general en scrum estos documentos se llaman **user stories**. El **product owner** prioriza las **user stories**, les asigna un valor y las agrega al **product backlog**.

Una de las críticas más fuertes que tiene el método scrum concierne con el rol del *product owner*: el hecho de que todo ese concimiento y responsabilidades recaiga en una sola persona es irreal e inaplicable. En la realidad, no existe una sola persona que concentre el rol del *product owner*, lo que redundaría en problemas con los que tiene que lidiar el equipo de desarrollo, tales como información cruzada, retrasos en obtención de información.

### Scrum Master

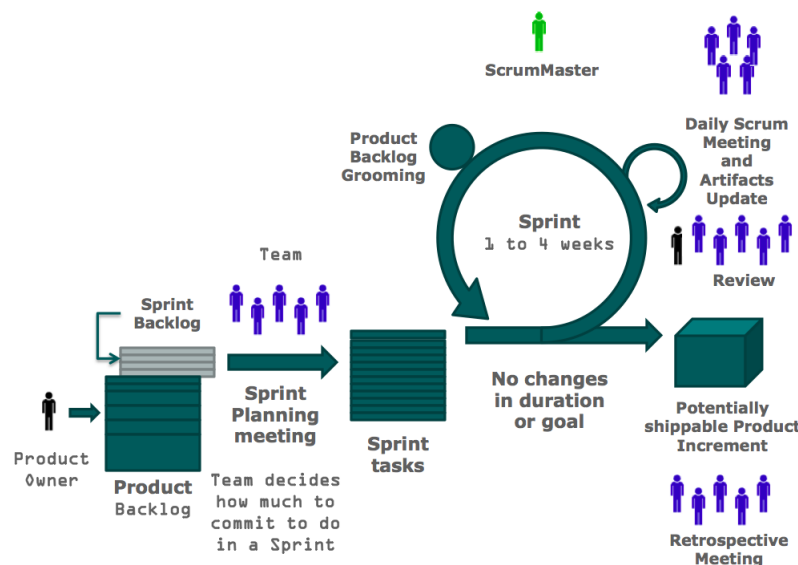
La persona con rol de *scrum master* es la que actúa bajo una suerte de líder facilitador o *coach* del equipo: su trabajo involucra remover cualquier impedimento que impida o retrase la productividad del equipo. Además, se ocupa de que se sigan los lineamientos de scrum durante el desarrollo. No se ocupa del manejo de personal ni de organizar al equipo de desarrollo. Concentra lo poco que quedó de los roles de gestión de los métodos tradicionales.

### Equipo

El **equipo** en scrum consiste de un grupo **auto-organizado** de entre 5 y 9 individuos que tiene la responsabilidad de entregar el producto. Este equipo no consta sólo de programadores sino que es interdisciplinario: involucra programadores, analistas, diseñadores, especialistas en bases de datos, en interfaz de usuario, testers, etc.

El rol de equipo también es objeto de críticas al método scrum. Se requiere que todos los individuos del equipo tengan un nivel de seniority elevado y parejo.

## 5.3 Diagrama general



## 5.4 Producto y Product backlog

Se define el **producto** como el resultado del proceso de desarrollo definido por el product owner. Para obtenerlo producto, el equipo parte del **product backlog**: una lista de items que describen funcionalidades y características que el proyecto debe tener:

- Involucra cosas pedidas por el usuario así como cosas propuestas por el equipo.
- Debe expresar los requerimientos funcionales y no funcionales.

- Es conveniente que incluya las características internas, no visibles al usuario (arquitectura, escalabilidad, uso de herramientas de desarrollo, etc).
- Pueden agregarse o quitarse ítems, o cambiar las estimaciones o la prioridad en cualquier momento.
- No hay un formato definido para los ítems – suelen ser *user stories*.
- Siempre hay un sólo *product backlog*.

Cada uno de los ítems del *product backlog* consta de:

- **Estado.** Puede ser:
  - Terminado: ya se hizo en un *sprint* anterior.
  - En el *sprint*: se está haciendo en este *sprint*.
  - Listo: está listo para agregarse a un *sprint*.
  - No listo: no puede agregarse a un *sprint* aún (por ejemplo, falta alguna dependencia).
- **Valor.** Representa cuánto estima el usuario que esa funcionalidad vale para el proyecto.
- **Esfuerzo.** Representa cuánto estima el equipo que va a costar hacer esa funcionalidad. Para determinar esfuerzo no se usa una escala absoluta sino relativa: se asigna 1 a la tarea más simple y se escala con eso como base (si una tarea tiene dificultad 3 es porque es 3 veces más difícil que la más fácil).
- **Prioridad.** Es una función del valor y del esfuerzo, usado para ordenar los ítems en el *product backlog*.

Los valores y los esfuerzos suelen estar limitados y se utiliza la serie de Fibonacci para asignarles valor. Particularmente para el esfuerzo se busca consenso en el equipo.

### Ejemplo de product backlog

ID	Item	Prio	Value	Effort	Status
1	As a buyer, I want to place a book in a shopping cart - <a href="#">Details</a>	1	7	5	Done
2	As a buyer, I want to remove a book from the shopping cart - <a href="#">Details</a>	2	5	2	Done
5	Improve transaction performance - <a href="#">Details</a>	3	4	13	In Sprint
7	Investigate solutions for speeding up credit card validation	4	4	20	In Sprint
6	Upgrade all servers to Apache 2.2.3	5	2	13	Ready
8	Diagnose and fix order processing script errors (JIRA SC-234)	6	3	3	Ready
3	As a shopper, I want to create and save a wish list - <a href="#">Details</a>	7	2	40	Not Ready
4	As a shopper, I want to add items to my wish list - <a href="#">Details</a>	8	2	20	Not Ready

## 5.5 User Stories

Una **user story** es un pequeño documento que especifica qué hay que hacer, evitando especificar cómo hacerlo. Estructuralmente, es similar a un caso de uso:

“As a [user role] I want to [goal] so that [benefit]”

A las *user stories* pueden agregarse detalles breves así como información adicional que ayude a definir el contexto. Además, es conveniente incluir los **criterios de aceptación** para esa *user story*: conjunto de condiciones que, si se satisfacen, esa *user story* se considera cumplida.

Uno de los principales beneficios de las *user stories* es que ayudan a pensar desde la perspectiva del usuario.

### 5.5.1 INVEST

Es conveniente que las *user stories* del *product backlog* cumplan una serie de características:

- **Independent.** Deben ser tan independiente de otras stories como sea posible
- **Negotiable.** Demasiados detalles en una *story* inhiben la comunicación Equipo-Owner-Cliente.
- **Valuable.** Deben tener valor para el usuario y/o para quien paga el desarrollo.
- **Estimable.** Legibles para el equipo y no demasiado grandes.
- **Small.** Menos de 3 semanas-hombre.
- **Testable.** Los criterios de aceptación deben ser tan objetivos como sea posible.

### 5.5.2 Ejemplo de user story

**#1 - As a buyer, I want to place a book in a shopping cart so I can choose the books that I will buy**

**Acceptance criteria:**

1. User can select a book from the catalog
2. User can search the catalog by author or title.
3. If the book is already in the shopping cart, its quantity is increased.
4. If the book is not already in the shopping cart, it is added with a quantity of one.
5. After the operation, the shopping cart is displayed.

**Effort: 5**

**Value: 7**

## 5.6 Inicio de un proyecto

Para iniciar un proyecto con scrum, se debe seguir la siguiente lista de pasos:

- Conseguir el apoyo del sponsor para usar scrum.
- Capacitar en Scrum a todos los involucrados (*product owner*, equipo, cliente final, etc).
- Planificación:
  - Definir la duración inicial de los *sprints* (1 a 4 semanas).
  - El *product owner* debe definir la visión inicial del producto.
  - El *product owner* debe crear el *product backlog* inicial y priorizar los ítems iniciales (al menos como para dos *sprints*).
    - \* Los ítems para los dos próximos *sprints* deben cumplir con INVEST.
    - \* Los ítems menos prioritarios pueden ser sólo un título (epic).
  - El equipo debe estimar los ítems más prioritarios.
    - \* La estimación suele hacerse en forma relativa, en *story points*.
  - Definición de los releases previstos y la funcionalidad que se espera entregar en cada uno.
  - Evaluación de riesgos.

## 5.7 Sprints

Un *sprint* (o iteración) es una unidad básica de desarrollo. Por el principio de *time boxing* de los métodos ágiles, consta de una longitud fija de tiempo, que no se extiende por ninguna circunstancia (incluso si el trabajo correspondiente a esa iteración no fue concluido).

Antes de comenzar cada iteración, el equipo decide qué elementos del *product backlog* entrarán en función de, entre otras cosas, la prioridad de los mismos. Es conveniente que cada *sprint* tenga un objetivo particular (por ejemplo, “implementación básica de validador de datos”). En general, este objetivo y un “alcance deseado” del sprint son presentados por el *product owner* y se da una etapa de negociación en la que el equipo, *scrum master*, *product owner* y *stakeholders* debaten si esas stories van a entrar o no. El equipo realiza una estimación de cuántos puntos de esfuerzo puede cumplir por iteración. Esta medida se llama **velocity**. Las *stories* que entraron en la iteración forman la *sprint list*.

Concluido esto, el equipo se reúne (junto con el *scrum master* y el *product owner* para responder dudas) para realizar el **plan detallado** del *sprint*, que consiste en descomponer cada *user story* en **tareas**. A cada tarea se le estima una cantidad de horas que tomará su realización. Como regla general, una tarea no puede tardar más de 8hs (una jornada laboral). Si eso ocurre, se la subdivide en tareas más pequeñas. A esta reunión se la llama **Sprint Planning Meeting**.

Durante las iteraciones, el equipo se reúne diariamente en una *daily scrum meeting*, donde se realiza un seguimiento del progreso. Se comenta qué se estuvo haciendo, a qué paso se viene avanzando, qué se planea hacer hasta la reunión siguiente, si hubo algún problema o retraso, etc.

Una vez que termina cada iteración, el equipo revisa los resultados con el *product owner* y los *stakeholders* y hace una demo. El resultado de un *sprint* debería ser un incremento en la funcionalidad, integrado y testeado. Además, el equipo con el *scrum master* se reúnen a hacer una retrospectiva para ver cómo fue la metodología de trabajo, qué se puede mejorar, qué se puede cambiar, etc.

Esta metodología con sus muchas reuniones colaboran mucho con la **visibilidad** de scrum.

## 5.8 Herramientas

### 5.8.1 Pizarra

El método scrum utiliza una **pizarra** para agrupar las tareas de la iteración actual de acuerdo con su estado:

- *Not Yet Started.*
- *In Progress.*
- *Completed.*
- *Blocked.*

A la hora de actualizar una tarea en la pizarra conviene que se haga inmediatamente. Cada tarea lleva su estimación de horas restantes, que debe ser actualizada antes o durante la reunión diaria.

La pizarra no es un elemento estático: las horas restantes de una tarea pueden crecer, así como la cantidad de tareas. Si surge una tarea inesperada se agrega a la pizarra, con su estimación. Nadie debe trabajar en tareas que no estén en la pizarra.

### 5.8.2 Mediciones

Para medir cuán productiva una iteración, el método scrum propone dos gráficos:

#### Product burndown chart

Es un gráfico de días que faltan en función de la fecha. Da una indicación de que tan rápido el equipo está terminando los requerimientos del *product backlog* en cada *sprint*.

#### Story burndown chart

Es un gráfico de horas restantes en función de días del *sprint*. Da una indicación de que tan rápido el equipo está terminando *stories* o “bajando horas” en un *sprint*. Muestran comportamientos disfuncionales.

## 5.9 Problemas de scrum

- Simplificación del *product owner*.
- Requiere un *product owner* involucrado durante todo el proceso de desarrollo.
- Se asume que todos los programadores tienen alto *seniority*.

## 6 Gestión de proyectos

Cuando un software crece, existen dos tipos de desafíos:

- De gestión: complicaciones en la logística del trabajo, tales como coordinación entre equipos, colaboración, diálogo, etc.
- Técnicos: complicaciones de escalabilidad, seguridad, portabilidad, soporte, etc.

### 6.1 ¿A qué se dedica un gerente?

Se ocupa de comunicar a las personas. Tiene un rol de coordinación. Entre la gente no se pueden comunicar, porque son muchos. Además, ayuda a facilitar el trabajo de la gente. Toma decisiones de rumbo. Determina algún curso de acción. Planificación.

En general, las funciones de gerenciamiento involucran:

- **Planificar:** es la fijación de objetivos y predeterminación de un curso de acción para lograrlos.
- **Organizar:** es la asignación de objetivos o tareas a roles.
- **Staffing:** asignación de personas concretas a cada uno de los roles que acabo de definir en la organización.
- **Liderar:** es el conjunto de tareas que tiene como objetivo crear un clima de trabajo en el cual la gente pueda hacer sus tareas motivada, en equipo y sin inconvenientes.
- **Controlar:** tiene que ver con el delta entre la realidad y el plan. Se debe analizar si ese delta es muy grande, considerar acciones correctivas (tales como reasignación de recursos, cambio de plan, de *staffing*, etc).

#### 6.1.1 Identificación de Stakeholders

Otra de las tareas clave de un gerente es la identificación de *stakeholders*. El gerente debe claramente identificar:

- Para quién se desarrollará el producto.
- Quién lo pagará.
- Quién lo usará.
- Quién es el factor de decisión esencial.
- Quién tiene el *know-how* (conocimiento de dominio, de formas, etc).
- Quién (y cómo) aceptará el producto terminado.

Algunos *stakeholders* clave son:

- Sponsor: es el que impulsa y le interesa que el proyecto avance.
- Lider usuario: es el que tiene más *know-how*.
- Usuarios directos e indirectos.

### 6.1.2 Determinación de factores críticos

Otra de las tareas de un gerente es la determinación de **factores críticos**. Es necesario determinar qué cosa es importante para el proyecto dado:

*“Puede ser bueno, lo puedo entregar rápido o puede ser barato. Elija dos.”*

En general podemos hablar de cinco dimensiones de la calidad en un proyecto de software:

- Funcionalidad.
- Calidad.
- Recursos.
- Costo.
- Plazo.

Cada uno de estos puede ser un *driver* (motivador), restricción (cota) o tener grados de libertad. Una forma de graficar esto es asignar un puntaje a cada uno de estas 5 dimensiones y graficarlo en un pentágono:

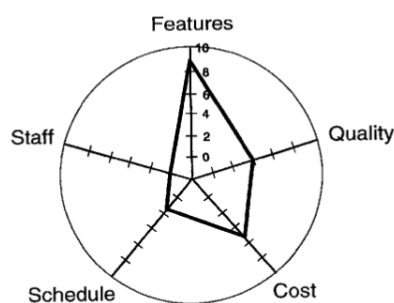


Figure 1: Sistema de información interno

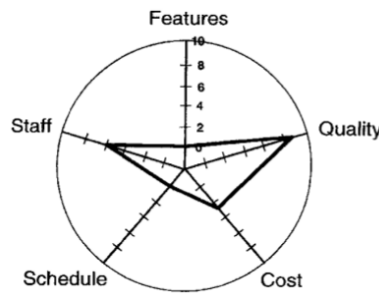


Figure 2: Aplicación comercial competitiva

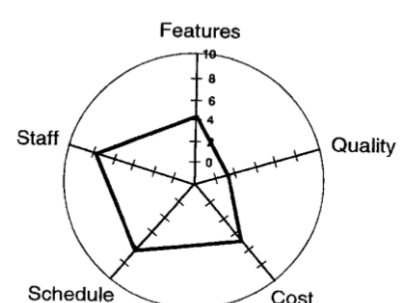


Figure 3: “Quality Driven”

### 6.1.3 Identificación preliminar de requerimientos

Otra de las tareas de un gerente es la identificación preliminar de requerimientos. Es un paso necesario para hacer la planificación, dado que inicialmente se usan para la estimación. Este paso involucra no solo requerimientos funcionales, sino también **atributos de calidad**.

### 6.1.4 Otros

Otras tareas importantes a realizar antes de comenzar un proyecto involucran:

- Elección de un modelo de ciclo de vida.
- Definición de un proceso para el proyecto.
- Definición de relación gente / duración y curva de *staffing* (cuidado con el “mito hombre hora”<sup>1</sup> y la “zona imposible”<sup>2</sup>).

<sup>1</sup>El **mito hora hombre** afirma que es falso que las horas y los hombres son intercambiables. *Por muchas mujeres que ponga a la tarea, no voy a tener un bebé en menos de 9 meses.*

<sup>2</sup>La **zona imposible** es el momento en el cual, por más que agregue más gente, no puedo comprimir más el tiempo del proyecto.

## 6.2 WBS

Un (WBS) (*Work Breakdown Structure*) es un gráfico que muestra la división de un proyecto o el producto en partes más pequeñas y manejables, hasta el nivel en que será ejecutado el control. Es el primer paso del armado de un cronograma.

Para realizar un WBS, es necesario, en primer lugar, definir su propósito (ej: poder estimar o asignar tareas) e identificar el nodo raíz. Luego, se subdivide cada componente en  $7 \pm 2$  subcomponentes hasta que se cumpla con el propósito.

### 6.2.1 Tipos de WBS

Existen tres tipos de WBS:

#### WBS de proceso

- La raíz identifica el nombre del proyecto.
- El segundo nivel identifica elementos mayores - Planificación, organización, análisis de requerimientos, diseño, iteraciones, etc.
- Partición de un proceso en subprocesos hasta obtener tareas individuales (1 o 2 personas) a desarrollar en poco tiempo (1 a 2 semanas).

#### WBS de producto

- Altamente relacionado con la arquitectura del producto.
- Identifica componentes e interfaces del producto.
- Identifica hardware, software y datos.
- La raíz identifica el nombre del producto.
- Los otros elementos son ítems discretos e identificables de hardware, software y datos.

#### WBS híbrido

- Combina elementos de los dos tipos anteriores.
- La raíz es un proceso, alternando elementos de proceso y producto.
- La idea es que los procesos producen productos y los subproductos requieren procesos para su desarrollo.
- Utilizado por managers que quieren priorizar la estimación y control precisos de cada elemento de producto.

### 6.2.2 Dependencias

Para poder optimizar la asignación de recursos es necesario entender las **dependencias** entre las tareas. Una dependencia es una relación entre dos o más tareas. Esta relación puede ser:

- Fin a comienzo.
- Fin a fin.
- Comienzo a fin.
- Comienzo a comienzo.

A la hora de armar un plan, la fecha de comienzo de tareas debe ser dinámicamente establecida en función de sus dependencias.

#### Tareas especiales

Se usa el recurso de **tarea hito**: es una tarea que tiene duración 0 que tiene un entregable asociado. Se utiliza como punto de control donde confluyen muchas tareas. Suelen tener alguna actividad relacionada.

Además, es necesario considerar e incluir **puntos de revisión y ajuste**, que suelen generar ciclos de tareas.

#### Dependencias externas

Es esencial considerar no sólo dependencias entre tareas internas al proyecto sino también dependencias con otros proyectos. Esto involucra no sólo tareas técnicas, sino también de negocios.

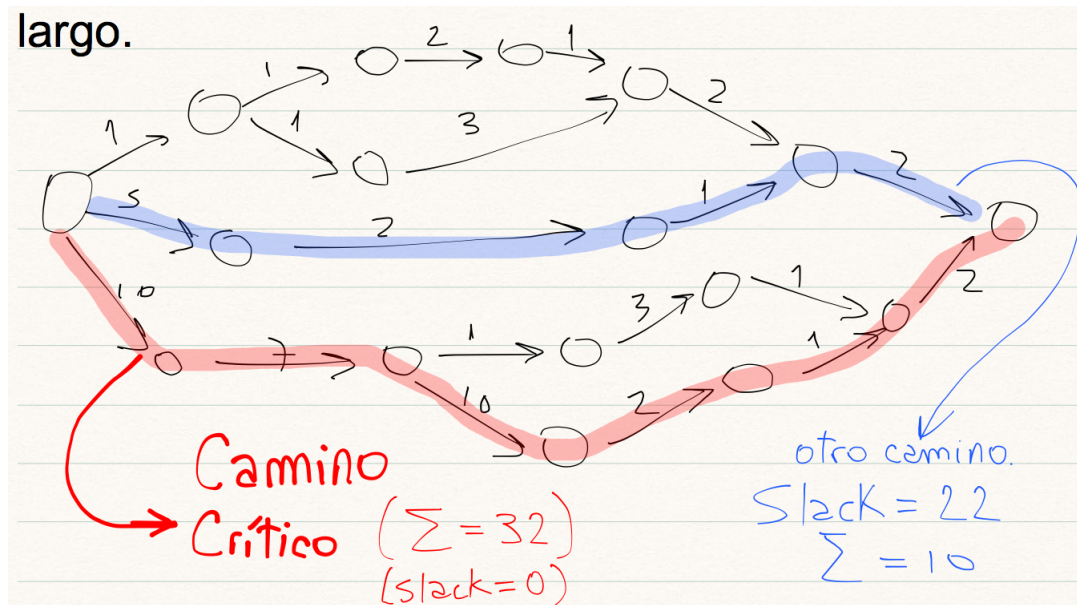
Además, no sólo hay que tener en cuenta proyectos “propios” sino también proyectos existentes en la organización con impacto en el propio.

Al final se agregan dependencias por contención de recursos: no tiene que ver con la esencia de las tareas, sino con que no tengo gente para hacer todas esas tareas en paralelo.

### 6.2.3 Camino crítico

Un **camino crítico** en un WBS es una secuencia de tareas en el DAG del proyecto cuyo atraso provoca atrasos en el fin del proyecto.

Toda camino tiene asociado un **slack** (margen): una cantidad de tiempo que se puede atrasar esa rama sin que se atrase todo el proyecto. Un camino es crítico si su *slack* es 0.



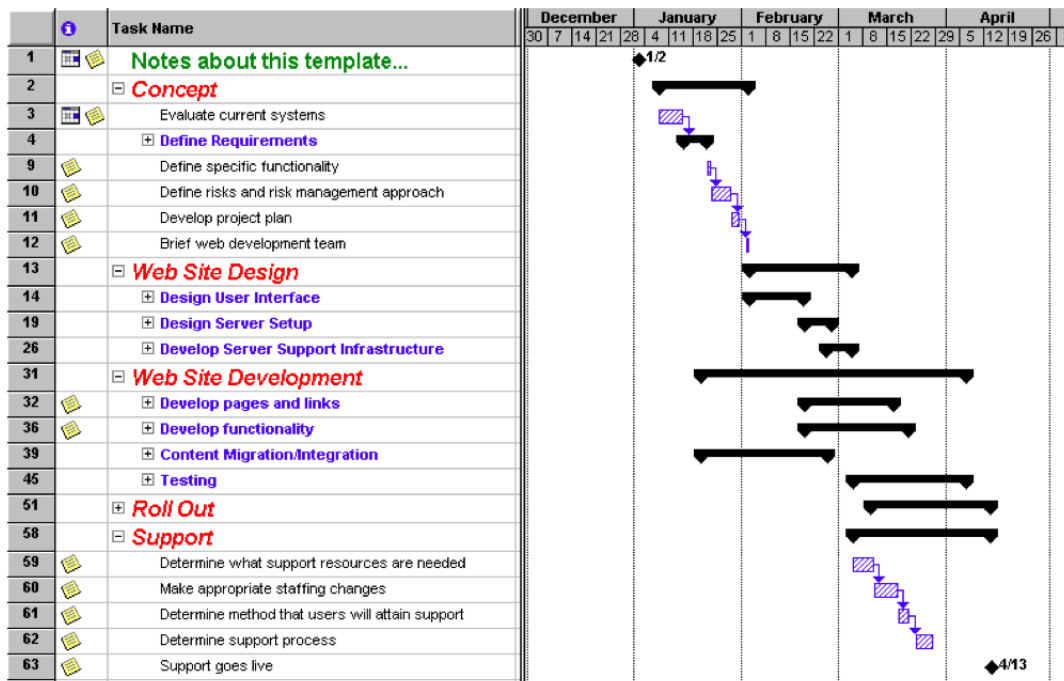
ewline

Es importante tener en cuenta los caminos críticos porque, lógicamente, es donde debe ir el mayor esfuerzo, más recursos, más atención, etc. Es lo que tengo que acortar si quiero acortar todo el proyecto. Además es crucial en la estimación: un análisis del plan para comprimir cronogramas debe comenzar por las tareas críticas.

El **lag** es el atributo de una arista que determina el tiempo mínimo que tiene que pasar entre que termino una tarea y puedo empezar la siguiente.

#### 6.2.4 Diagrama de Gantt

Un **diagrama de Gantt** es un gráfico que me muestra el ordenamiento de las tareas a lo largo del tiempo. Muestra tareas con responsables, fechas, secuencia de ejecución y costos directos. Sirve fundamentalmente como referencia para la ejecución y control del proyecto. Para presentaciones se suelen usar sólo diagramas de hitos o de componentes de alto nivel, sin incluir la descripción detallada de actividades.



### 6.2.5 Otras definiciones

- **Línea de base:** Es una versión estable de mi plan que se usa como base para hacer el seguimiento. Es contra lo que voy a comparar la planificación actual.
- **Programación por valor acumulado:** tiene que ver con la medición del progreso. Para medir avance se usa la fórmula de valor acumulado: definición de pesos a los entregables de los hitos, para medir avance (no esfuerzo). Informalmente, se asocian entregables a cada hito. Esos entregables tienen un peso que voy acumulando a medida que voy logrando.

## 6.3 Gestión de riesgos

En el contexto de gestión de proyectos, se define un **riesgo** como un problema que todavía no ocurrió. Un **problema** es un riesgo que se manifestó.

Los riesgos tratan sobre posibles eventos en el futuro y están caracterizados por:

- **Probabilidad** de que el riesgo se manifieste.
- **Impacto** si lo hace.

Se define la **exposición** a un riesgo como  $probabilidad \times impacto$ . Uno de los principales problemas de la ingeniería del software es que no hay forma exacta de cuantificar estas medidas. Se suelen trabajar “artesanalmente”.

Estas dos caracterizaciones definen las dos principales formas de atacar un riesgo: reducir la probabilidad de que ocurra o su impacto negativo si ocurre.

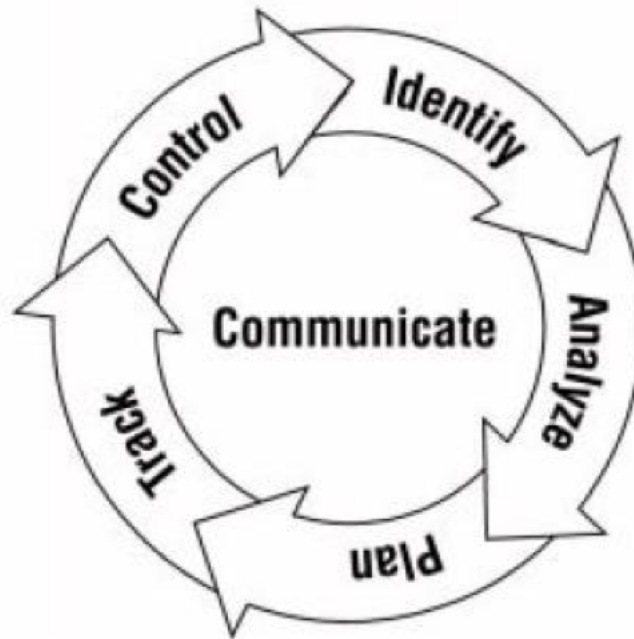
Una **fuentes de riesgo** es algo que me indica que un riesgo está presente.

### 6.3.1 Sistematización

Para sistematizar el tratamiento de riesgos, la IEEE propone realizar ciclos de:

- Identificación de riesgo.

- Análisis de riesgo.
- Planificación.
- Seguimiento del plan.
- Aplicación de medidas de control.



### 6.3.2 Identificación de riesgos

Existen numerosos métodos para identificar riesgos:

- Brainstorms.
- Reporte periódico de riesgos.
- Cuestionario de identificación taxonómica. Es un cuestionario de 194 preguntas si/no. Es obsoleto.
- Reportes voluntarios de riesgos.
- Listas de riesgos comunes,

### 6.3.3 Documentación

Dado que los riesgos son algo inevitable e inherente del desarrollo de software, es importante que queden bien documentados. Para esto se recomienda usar la representación de Gluch:

“**Dado que** [condiciones], **entonces, posiblemente,** [consecuencias]”

ewline

Para categorizar los riesgos se los ordena en una matriz de probabilidad  $\times$  impacto llamada **matriz de magnitudes**:

<div>Probabilidad</div> <div>Severidad</div>	Muy Probable	Probable	Poco Probable
Crítica	Alta		
Media		Media	
Marginal			Baja

Otro aspecto de la documentación involucra la definición de prioridades. Una vez categorizados los riesgos en la matriz de magnitudes, ordenamos los riesgos según la matriz de magnitudes, ignorando los riesgos de baja magnitud. Para los riesgos de igual magnitud ponemos primero los que requieran acciones correctivas con más urgencia y/o impacto.

Siempre tratamos de quedarnos con los 5 o 10 riesgos con mayor exposición.

#### 6.3.4 Planificación

Un **plan de contingencia** es el conjunto de medidas que han de tomarse si un riesgo se manifiesta. Es muy importante que los planes de contingencia estén bien documentados incluyendo, entre otros en qué circunstancias y quién aplica el plan de contingencia. Además, debe ser implementable.

### 6.4 Plan de gestión de proyecto

Armar un plan de proyecto es más complejo que simplemente armar un cronograma. Incluye cosas como:

- Organización del proyecto.
- Definición de la estructura interna.
- Modelo de ciclo de vida
- Infraestructura.
- Roles y responsabilidades.
- Gestión de configuración.
- QA.
- Gestión de calidad.
- Comienzo del proyecto.
- Plan de trabajo / cronograma.
- Plan de control.
- Gestión de riesgos.

- Cierre del proyecto.
- Organización de planes de proceso.

El **plan de gestión de proyecto** es un documento consistente y coherente para guiar la ejecución y el control del proyecto. Su uso está reglamentado en un estándar de la **IEEE**.

## 7 Atributos de calidad

La **ingeniería de requerimientos** es una forma disciplinada y sistemática de llegar desde las necesidades de los usuarios hasta una especificación. Para definir correctamente una arquitectura, es imprescindible conocer tres elementos del software:

- Funcionalidad “de negocio” (requerimientos funcionales).
- Atributos de calidad (requerimientos no-funcionales).
- Restricciones.

Un proyecto de software no se compone enteramente de su funcionalidad **de negocio**. También existen los **atributos de calidad** (“ilities”), que referencian características específicas que debe tener el sistema. Antiguamente se llamaban requerimientos no-funcionales. Existen numerosos atributos de calidad, los tratados en la materia son:

- Performance.
- Disponibilidad.
- Testeabilidad.
- Modificabilidad.
- Usabilidad
- Seguridad.

(Sugerencia: un acrónimo para recordarlos es **PeDiTe MoUsSe**.)

Es importante notar que estos problemas no son completamente independientes, sino que se afectan mucho entre si. Es común que una medida que se tome para mejorar uno de estos atributos de estos de calidad vaya en detrimento de otra. Por ejemplo, una mejora en seguridad suele empeorar la usabilidad de un sistema.

Según la **IEEE**, la **calidad** “*La calidad de un software es el grado en el que un software tiene una combinación deseada de atributos*”.

Para atacar los atributos de calidad, se usan **tácticas** de arquitectura<sup>12</sup>. Estos no son objetivos en si mismos, sino formas de alcanzar atributos de calidad.

### 7.1 Generales vs Concretos

#### Generales:

- Son independientes de un sistema específico y por lo tanto pueden, potencialmente, pertenecer a cualquier sistema.
- Son una guía para saber qué preguntar o pensar en escenarios candidatos.
- Están ordenados por atributo de calidad.

#### Concretos

- Son específicos a un sistema, instanciando cada uno de sus atributos.

## 7.2 Los 6 atributos

### 7.2.1 Disponibilidad

El atributo de calidad de **disponibilidad** está relacionado con las fallas (*failures*) que puede tener el sistema y sus consecuencias asociadas. Se considera que ocurre una **falla** (*failure*) un sistema no entrega más un servicio de acuerdo con su especificación. Estas fallas (*failures*) son observables por los usuarios.

La disponibilidad es la probabilidad de que un sistema esté disponible cuando se lo necesite. Un sistema se dice **tolerante a fallas** si puede seguir operando en presencia de fallas.

Es importante entender la distinción entre los siguientes conceptos:

- **Fault**: un paso, proceso o definición incorrecta en un sistema que causa que el programa no responda en la forma esperada. Es una debilidad inherente al proceso de desarrollo de software que redunde en un *failure*.
- **Failure**: la incapacidad de un sistema o componente de realizar sus funciones requeridas por su especificación.
- **Error**: una diferencia entre el valor observado o medido por un cómputo y el valor “correcto”, especificado por la teoría.

Se define el **tiempo de reparación** como el tiempo hasta que la falla no es más observable.

#### Tipos de fallas

- Crash failure: el sistema funciona perfectamente hasta que ocurre un error inesperado.
- Omission failure: el sistema falla recibiendo requerimientos, enviando o recibiendo mensajes.
- Timing failure: el sistema responde fuera de los tiempos esperados.
- Response failure: respuesta del sistema incorrecta o se desvía del flujo de control correcto.

#### Medidas para hacer un sistema más tolerante a fallas

- *Voters*: un “voter” recibe entradas y replica en la salida el valor que tenga la mayoría de sus entradas.
- Redundancia de voters.
- Sistemas duplex: poner  $2n + 1$  procesadores que computen lo mismo y voten.

#### Ejemplo

- “En zonas donde la cobertura 3G sea débil o inexistente, se debe mantener la conexión entre las Terminales de Cobro móviles y el Sistema Central utilizando paquetes GSM SMS”.
- “Si la comunicación entre las terminales de cobro y el Sistema Central se pierde, o los tiempos de transmisión son prohibitivos, las terminales de cobro deben seguir funcionando en modo offline, de forma transparente al usuario”.

### 7.2.2 Modificabilidad

El atributo de **modificabilidad** hace referencia al costo de realizar cambios en el sistema. Este costo no sólo referencia el costo económico de los cambios, sino también tiempo y esfuerzo invertido.

Estos cambios pueden ser:

- Funcionalidad.
- Plataforma.
- Otros atributos de calidad.
- Interfaces.

Hacer un cambio involucra:

- Especificarlo.
- Diseñarlo.
- Implementarlo.
- Probarlo.
- Ponerlo en producción.

### Ejemplo

- “Si bien existen varias maneras para realizar la recarga de una tarjeta de proximidad, es probable que en el futuro se desee aceptar nuevas formas de pago, por ejemplo, vía web. Es por ello que se requiere tener un sistema fácil de extender a nuevas formas de pago sin que esto impacte demasiado y sea rápido de implementar”.
- “Se pretende, en un futuro, implementar un sistema tarifario basado en zonas, en el cual el costo del pasaje este en relación con la cantidad de zonas que el pasajero atraviesa y no respecto de la cantidad de transportes que utilice. Por lo tanto, se espera que el sistema sea fácil de modificar para aceptar nuevas formas de realizar el cobro de los pasajes”.

### 7.2.3 Performance

El atributo de **performance** está relacionado con el tiempo que le lleva al sistema responder a un evento (interrupción, mensaje, pedido, etc) que ocurre. Es extremadamente difícil de expresar porque depende de innumerables factores, tales como:

- **Latencia:** tiempo entre la llegada del estímulo y el inicio de la respuesta del sistema.
- **Jitter:** variación en la latencia.
- **Deadlines:** límites de tiempo para un proceso.
- **Throughput:** cantidad de transacciones que el sistema puede procesar en un período de tiempo.
- Eventos no procesados.

## Ejemplo

- “Se requiere que el sistema sea capaz de realizar las operaciones de autenticación y cobro en a lo sumo 1 segundo ya que, de otro modo, no sería bien recibido los usuarios, ni los choferes de colectivos. Esto es especialmente crítico en horas pico”.

### 7.2.4 Seguridad

El atributo de **seguridad** predica sobre la habilidad de un sistema para resistir usos no autorizados y seguir proveyendo sus servicios a usuarios legítimos. Involucra:

- **Confidencialidad:** se garantiza que la información sea accesible sólo por aquellas personas autorizadas.
- **Integridad:** se salvaguarda la exactitud y totalidad de la información y los métodos de procesamiento. La integridad incluye: la integridad de los datos (el contenido) y el origen de los mismos.
  - Integridad de datos: nadie altere el contenido.
  - Integridad de origen (*nonrepudiation*): el origen de los datos sea cierto.
- **Disponibilidad:** se garantiza que los usuarios autorizados tengan acceso a la información y a los recursos relacionados con la misma, toda vez que lo requieran.

Se considera parte de integridad la **auditabilidad**, que habla de la habilidad de un sistema para hacer un seguimiento de actividades realizadas.

## Ejemplo

- “Es crítico que nadie fuera del Sistema Central y el Banco Nación conozca ninguno de los datos de los usuarios, sus compras realiza, recargas efectuadas, etc”.
- “Es también muy importante que los datos (por ej. de una recarga) lleguen de forma correcta, y sin ningún tipo de modificación por factores externos como la forma de transmisión”.
- “Deben de detectarse potenciales diferencias malintencionadas entre los saldos almacenados de forma local en las tarjetas y los valores guardados para las mismas en el Sistema Central. Nadie debería viajar gratis”.
- “El Sistema Central debe mantener un registro completo de absolutamente todos los viajes que se realizaron”.

### 7.2.5 Usabilidad

La **usabilidad** está relacionada con la facilidad con la cual un usuario puede cumplir una tarea o utilizar un servicio ofrecido por el sistema y el tipo de soporte que provee el sistema.

Involucra cosas como:

- Aprender la funcionalidad del sistema.
- Usar el sistema eficientemente.
- Minimizar el impacto de los errores.
- Adaptar el sistema a las necesidades de los usuarios.
- Aumentar confianza y satisfacción.

## Testeabilidad

- “Dado que en los colectivos será el propio pasajero quien determine el valor del pasaje mediante una interfaz de tipo touchscreen, se desea que la misma sea intuitiva y permita realizar dicha operación de manera rápida y sencilla en la mayoría de los casos”.
- “Se quiere que el sistema de información de tarifas, rutas, y tiempos sea accesible para personas con discapacidades visuales”.

### 7.2.6 Testeabilidad

La **testeabilidad** es la posibilidad de ver el *estado interno* de la aplicación.

## 7.3 Especificación

Uno de los principales problemas de los atributos de calidad es que suelen estar pobremente o nulamente especificados. No se suelen analizar sus dependencias, ni se les da importancia a la hora de hacer una especificación formal. Sin embargo, su nivel de importancia puede llegar a ser muy alto (varía dependiendo del proyecto particular).

Una forma de especificar un atributo de calidad es mediante un **QAS** (*Quality Attribute Scenario*). Una QAS es una tupla:

- **Fuente:** interna o externa.
- **Estímulo:** condición que debe ser tenida en cuenta al llegar al sistema Entorno: condiciones en las cuales ocurre el estímulo.
- **Artifact:** el sistema o partes de él afectadas por el estímulo
- **Response:** qué hace el sistema ante la llegada del estímulo.
- **Response measure:** cuantificación de un atributo de la respuesta.

### 7.3.1 Ejemplos

### 7.3.2 Disponibilidad

“Si la comunicación entre una Terminal de Cobro móvil y el Sistema Central se pierde, o los tiempos de transmisión son prohibitivos, la Terminal de Cobro debe de seguir funcionando en modo offline, de forma transparente al usuario”.

- Fuente: Interno al sistema.
- Estímulo: Tiempo promedio transacciones mayor a 1 seg.
- Entorno: Operación normal.
- Artefacto: Terminal de Cobro.
- Respuesta: La terminal pasa a modo offline, utilizando la información local almacenada en las tarjetas y logueando toda operación realizada para que este lista cuando la conexión se restablezca.
- Medición: No se requieren acciones adicionales por parte del usuario, y no se ve afectada la performance ( $\leq 1$  seg. por trans.).

### 7.3.3 Modificabilidad

“Se pretende, en un futuro, implementar un sistema tarifario basado en zonas, en el cual el costo del pasaje este en relación con la cantidad de zonas que el pasajero atraviesa y no respecto de la cantidad de transportes que utilice. Por lo tanto, se espera que el sistema sea fácil de modificar para aceptar nuevas formas de realizar el cobro de los pasajes”.

- Fuente: Ministerio de Planificación.
- Estímulo: Agregar al sistema la división por zonas y basar el costo de los pasajes en las mismas.
- Entorno: En tiempo de diseño.
- Artefacto: Sistema.
- Respuesta: Cambio efectuado sin efectos secundarios, el pago se efectúa con la nueva modalidad de cobro.
- Medición de la respuesta: Se invierten menos de 550 horas hombre. La usabilidad no se ve afectada negativamente.

### 7.3.4 Performance

“Se requiere que el sistema sea capaz de realizar las operaciones de autenticación y cobro en a lo sumo 1 segundo ya que, de otro modo, no sería bien recibido por los usuarios, ni los choferes de colectivos. Esto es especialmente crítico en horas pico”.

- Fuente: Pasajero.
- Estímulo: Se aproxima o inserta una tarjeta para realizar el pago de un pasaje.
- Entorno: En operación normal y hora pico.
- Artefacto: Terminal de cobro.
- Respuesta: Se concreta la operación de la venta del pasaje.
- Medición de la respuesta: tiempo de respuesta total  $\leq 1$  seg.

### 7.3.5 Seguridad

“Es crítico que nadie fuera del Sistema Central y el Banco Nación conozca ninguno de los datos de los usuarios, sus compras realizadas, recargas efectuadas, etc”

- Fuente: Atacante externo.
- Estímulo: Captura un mensaje encriptado e intenta vulnerar la seguridad para obtener información confidencial de las transacciones de los usuarios.
- Entorno: Operación Normal.
- Artefacto: Módulos de encriptación de las terminales y el Sistema Central.
- Respuesta: Los datos no son accesibles al usuario en tiempos razonables (miles de años).
- Medición de la respuesta: (tiempo para “romper” el sistema, % de casos en los que el ataque no es exitoso).

### 7.3.6 Usabilidad

“Dado que en los colectivos será el propio pasajero quien determine el valor del pasaje mediante una interfaz de tipo touchscreen, se desea que la misma sea intuitiva y permita realizar dicha operación de manera rápida y sencilla en la mayoría de los casos”.

- Fuente: Pasajero.
- Estímulo: Selecciona pasaje que quiere adquirir.
- Entorno: Operación normal.
- Artefacto: Terminal de Cobro/Recarga, Interfaz gráfica.
- Respuesta: El pasajero realiza la operación satisfactoriamente.
- Medición de la respuesta: En test usabilidad, 95% de los pasajeros pueden realizar la operación, sin ningún tipo de ayuda de otra persona, la primera vez que se enfrentan al sistema, y en menos de 20 segundos.

## 7.4 Quality Attribute Workshops

Un *Quality Attribute Workshop* (QAW) es un método que relaciona los *stakeholders* de un sistema de manera temprana en el ciclo de vida para descubrir los atributos de calidad clave en un sistema de software. Sus pasos son:

- Presentación del método: describir el propósito del QAW y a los “stakeholders” (posición en la organización y rol en el sistema)
- Presentación del negocio / misión: presentación realizada por un representante de los stakeholders cuyo objetivo principal es entender la motivación para construir el sistema.
- Presentación del plan de arquitectura: un representante técnico presenta lo que se sepa sobre la arquitectura:
  - Planes y estrategias.
  - Requerimientos y restricciones clave.
  - Diagramas de alto nivel.
- Identificar drivers arquitectónicos: primera depuración de los resultados de los pasos anteriores. Involucra una clasificación en requerimientos, restricciones y atributos de calidad requeridos.
- Brainstorming de escenarios: consiste en buscar escenarios de los 3 tipos (caso de uso, crecimiento, exploratorios) y se los documenta como estímulo, entorno o respuesta.
- Consolidación de escenarios: los facilitadores agrupan escenarios similares
- Definición de prioridades de escenarios: cada stakeholders “vota” por hasta un 30% de los escenarios consolidados.
- Refinamiento de escenario: se refinan los 4 o 5 escenarios más votados.

## 8 UML

Hacia principios de los '90, coexistían distintas notaciones y enfoques para modelado de objetos. En ese contexto, **UML** (*Unified Modeling Language*) surgió como un estándar de modelado que combina modelos nuevos y existentes, e intenta cubrir todas las necesidades de especificación de un sistema de software.

**UML no es una metodología ni un proceso, es una notación estándar de modelado para capturar el conocimiento semántico de un problema y su solución.**

Un **proceso** es una descripción de actividades que deben realizarse en un determinado orden (en este caso para crear o modificar un sistema de software). Describe qué hacer, cómo hacerlo, cuándo hacerlo, qué roles deben hacerlo y el motivo por el que se hace. Debe ser: reproducible, definido, medible y mejorable.

## 9 Unified Process

El **unified process** es un marco de desarrollo de software que se caracteriza por estar dirigido por casos de uso, centrado en la arquitectura y por ser iterativo e incremental. Se centra en la **arquitectura**, pero está dirigido por casos de uso.

Es importante notar que **UP** es un *framework*: se puede (y debe) adaptar a las características de un proyecto.

Se define el concepto de **risk-based development**: los riesgos guían la elección de la funcionalidad.

### 9.0.1 Buenas prácticas

- Desarrollar iterativamente.
- Administrar los requerimientos.
- Usar arquitecturas de componentes.
- Modelar visualmente (UML)
- Verificar la calidad.
- Controlar los cambios.

### 9.1 Iteraciones

A diferencia de scrum, en **UP** el resultado de cada iteración es un sistema funcionando, aunque no necesariamente “potentially shippable”. Hay que tener cuidado con las “salidas a producción intermedias”.

Sin embargo, al igual que en scrum, hay aprendizaje entre las iteraciones y se aplica el concepto de *timed boxing* (entre 2 y 12 semanas). Además se busca el feedback del usuario.

Las iteraciones no son estrictamente secuenciales. Las disciplinas se superponen.

#### 9.1.1 Fases

- |  |   |                       |
|--|---|-----------------------|
| • Inicio del proyecto: definir contexto, factibilidad y objetivos. | } | Etapas de Ingeniería. |
| • Elaboración: funcionalidad (alcance) y arquitectura básica.      |   |                       |
| • Construcción: desarrollar el producto iterativamente.            | } | Etapas de Producción. |
| • Transición: liberar el producto para uso real.                   |   |                       |

### 9.1.2 Hitos

En UP, un **hito** es un punto de control para revisar el avance del proyecto. Tienen entregables asociados para evaluación.

Existen dos tipos de hitos:

- Principales al finalizar una fase (visión, arquitectura básica, capacidad inicial, producto final).
- Secundarios al finalizar una iteración.

Algunas definiciones:

- **Disciplinas:** organizan las actividades del proyecto. Cada disciplina de desarrollo genera modelos de UML (casos de uso, análisis, diseño, deployment, etc).
  - De desarrollo: requerimientos, análisis, arquitectura, diseño, implementación, pruebas, deployment.
  - De gestión: administración de riesgos, planificación y seguimiento, SCM (software configuration management), etc.
- **Artefactos:** Cualquier tipo de información generada por el equipo de desarrollo

En general el número de artefactos generados es muy grande. Luego, es preciso definir cuáles se harán en un desarrollo concreto:

- 

### 9.1.3 Fase de inyección

Los propósitos de la fase de inyección son:

- Establecer el “caso de negocio” para un sistema nuevo o mejoras a uno existente.
- Además se debe especificar el alcance del proyecto.

Lo que se espera obtener de la fase de inyección es:

- Visión de los requerimientos (10-20%).
- Caso de negocio.
  - Criterios de éxito.
  - Estimación de recursos requeridos.
  - Evaluación inicial de riesgos.

Su hito principal es: “Lifecycle objectives”.

### 9.1.4 Fase de elaboración

Los propósitos de la fase de elaboración son:

- Analizar el dominio del problema.
- Establecer una base sólida de arquitectura.
- Atacar principales riesgos.
- Desarrollar un plan completo.

Lo que se espera obtener de la fase de elaboración es:

- Modelo de dominio y casos de uso (80% completo).
- Arquitectura probada y documentada.
- Tener revisado el caso de negocio.
- Tener terminado el plan de desarrollo.

### 9.1.5 Fase de construcción

En construcción se especifica, desarrolla y prueba el producto de manera incremental.

### 9.1.6 Fase de transición

En transición se llevan a cabo todas las tareas necesarias para una salida a producción:

- Instalación.
- Configuración.
- Entrenamiento.
- Soporte.
- Mantenimiento.

No se deben confundir las iteraciones de transición con los ciclos de pruebas.

## 9.2 Ventajas y desventajas

		<u>Problemas:</u>
<u>Ventajas:</u>		
<ul style="list-style-type: none"><li>• Buenas Prácticas</li><li>• Casos de uso</li><li>• Iteraciones</li><li>• Arquitectura</li><li>• UML</li></ul>		<ul style="list-style-type: none"><li>• “Sopa de prácticas”</li><li>• Demasiado grande</li><li>• Adopción muy difícil</li><li>• “Nobody reads process books”</li><li>• Extensión difícil</li><li>• <i>Process gap</i>.</li></ul>

## 10 Arquitecturas

La **arquitectura de un sistema de software** define el sistema en términos de elementos e interacción entre ellos. Es como una descripción de “alto nivel” del diseño. Muestra correspondencia entre requerimientos y elementos del sistema construido.

Resuelve atributos de calidad en el nivel del sistema, como escalabilidad, flexibilidad, confiabilidad y performance

Informalmente, intenta plantear una respuesta al problema de construir un puente entre los requerimientos y las soluciones (diseño detallado / implementación).

Las arquitecturas son importantes porque son un paso clave para poder lograr el reuso a gran escala y ayudar a que la ingeniería de software empiece a asemejarse a otras disciplinas de la ingeniería. Además, facilita la comunicación temprana entre “stakeholders” y ayuda a tomar decisiones tempranas de diseño tales como:

- Define restricciones de implementación
- Afecta la estructura organizativa
- Posibilita o inhibe atributos de calidad de un sistema
- Facilita el razonamiento sobre cambios y su implementación
- Permite estimaciones más precisas

Además, la arquitectura hace posible la abstracción transferible de un sistema: los sistemas pueden ser contruidos a partir de elementos externos, lo que permite el “desarrollo basado en templates”.

## 10.1 Definiciones

Según Bass y Clemens, la **arquitectura de software de un sistema de computación es el conjunto de estructuras necesarias para razonar sobre el sistema. Comprende elementos de software, relaciones entre ellos y propiedades entre ambos.**

Un **estilo** (o patrón arquitectónico) es una descripción de tipos de relaciones y elementos, junto con restricciones sobre cómo deben usarse (ej. “client server”).

Una **arquitectura de referencia** es una división común de funcionalidad mapeada a elementos que cooperativamente implementan esa funcionalidad y flujos de datos entre ellos. Ej: uso de sensores y actuadores en aplicaciones de robótica.

Una arquitectura es **heterogénea** si resulta de la combinación de distintos estilos arquitectónicos.

## 10.2 Los tres principios fundamentales

- Toda aplicación tiene una arquitectura.
- Cada aplicación tiene al menos un arquitecto.
- La “arquitectura” **no** es una fase del desarrollo.

## 10.3 ¿Qué hace que una arquitectura sea buena?

- Producto de un único arquitecto o un pequeño grupo de arquitectos con un claro líder. **Integridad conceptual.**
- El equipo de arquitectura debe contar con requerimientos funcionales y atributos de calidad requeridos que sean claros.
- La arquitectura debe estar documentada.
- La arquitectura debe ser revisada por los *stakeholders*.
- Debe ser evaluada cuantitativamente antes de que sea tarde.
- Debe permitir una implementación incremental.
- Módulos bien definidos basados en el ocultamiento de la información.
- Interfaces claramente definidas.
- No dependiente de un único producto comercial.
- Usa un grupo pequeño y claro de patrones de interacción.

## 10.4 Documentación de una arquitectura

Elementos esenciales:

- Descripción de los requerimientos: contexto del negocio, “rationale” para el producto, dominio.
- Descripción del contexto: sistemas con quienes interactúa, interfaces externas.
- Uso de diagramas de arquitectura: con prosa y descripción de cajas y líneas.
- Consideración de restricciones de implementación (en la medida en que impactan la arquitectura).
- Explicación del diseño arquitectónico: como ataca los requerimientos y las restricciones de diseño. Alternativas.

Hay varios puntos clave a tener en cuenta para documentar una arquitectura:

- El documento está escrito para el lector, no quien lo escribe.
- Evitar ambigüedad.
  - Explicar la notación – claves.
  - Adjuntar información adicional, por ejemplo de interfaces.
- Usar una organización estándar (qué vistas, template a usar)
- Registrar los motivos (explicar el por qué de las decisiones tomadas).
- Mantenerla actualizada, pero no demasiado.
- Revisarla.

## 10.5 Arquitectura vs diseño

No hay consenso con respecto a la relación entre arquitectura y diseño. La visión más aceptada es que la arquitectura es una parte del diseño, que también involucra decisiones no arquitectónicas (que pueden convertirse en arquitectónicas más adelante, dependiendo de la visibilidad de sus módulos).

Recordemos que la definición de **diseño** es: “formar un plan o esquema en la mente para ejecución posterior”.

## 10.6 ADD

ADD (*Attribute Driven Design*) es un método para desarrollar una arquitectura que consiste en:

- Elegir el módulo a descomponer.
- Refinar el módulo:
  - Elegir drivers de arquitectura a partir de Escenarios de Atributos de Calidad y requerimientos funcionales.
  - Elegir un patrón arquitectónico que satisfaga los drivers.
  - Instanciar módulos y asignar funcionalidad / representar usando vistas.
  - Definir interfaces de módulos hijos.
  - Verificar y refinar casos de uso y escenarios.
- Iterar.

## 11 Estilos arquitectónicos

Un **estilo arquitectónico** es una colección de decisiones de arquitecturas que son aplicables en un contexto de desarrollo, se encuentran repetidamente en la práctica y tiene propiedades beneficiosas que permiten y justifican su reuso. Informalmente, es una descripción (de muy alto nivel y sin hablar de funcionalidad específica) de tipos de relaciones y elementos, junto con restricciones sobre cómo deben usarse.

Son el equivalente a los patrones de diseño, pero con ciertas diferencias. Los estilos arquitectónicos tienen en común con los patrones que ambos proveen un “lenguaje común” y una abstracción con la que se describen clases de sistemas. La principal diferencia es que un patrón es una solución a un problema recurrente, mientras que un estilo es más general y no requiere de un problema para su aparición.

Los estilos arquitectónicos tienen restricciones topológicas que determinan cómo se puede hacer la composición de los elementos que involucran (por ejemplo, “los elementos de un *layer* sólo se pueden comunicar con los de un *layer* inferior”) y provee semántica para esos elementos.

Es importante notar que un estilo arquitectónico **no define la funcionalidad de un sistema**. Es abstracto.

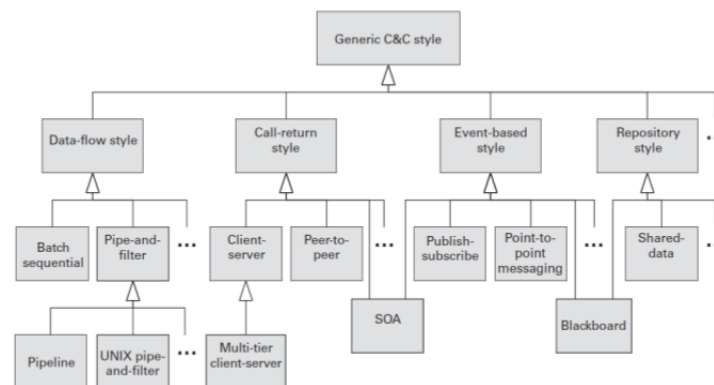
### 11.1 Ventajas

- Reuso de diseños: soluciones maduras aplicadas a problemas nuevos.
- Reuso de código: una parte importante del código que implementa la arquitectura puede pasarse de un sistema a otro.
- Comunicación.
- Portabilidad.

### 11.2 Taxonomía

La **taxonomía** es la ciencia de la clasificación. En este caso, se aplica una clasificación a los estilos arquitectónicos, agrupándolos por características comunes. No existe una única taxonomía.

Ejemplo:



### 11.3 Estilos

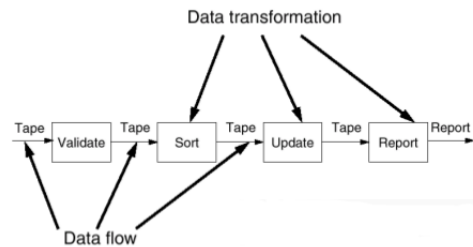
#### 11.3.1 Data Flow

En los estilos tipo **data flow**, la estructura del sistema está basada en transformaciones sucesivas a datos de *input*: los datos entran al sistema y fluyen a través de los componentes hasta su destino final.

Normalmente existe un componente de control que controla la ejecución del resto de los componentes. Hay dos subestilos:

### Batch sequential

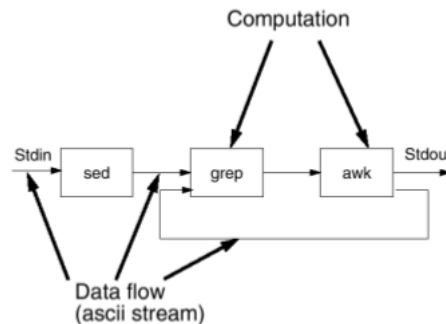
En **batch sequential**, cada paso se ejecuta hasta ser completado, y recién después puede comenzar el siguiente paso. Es usado en aplicaciones clásicas de procesamiento de datos.



### Pipe and filter

En **pipe and filter**, cada componente tiene *inputs* y *outputs*. Los componentes leen *streams* de datos de su *input* y producen *streams* de datos en sus *outputs* de forma continua.

- *Filters*: ejecutan las transformaciones
- *Pipes*: conector



### Ventajas y desventajas

#### Ventajas:

- Fácil de entender: la función del sistema es la composición de sus filtros.
- Facilidad de extensión – reuso – recambio de filtros.
- Posibilidad de ejecución concurrente.

#### Problemas:

- “Mentalidad batch”, difícil para aplicaciones interactivas.
- El orden de los filtros puede ser complejo.
- Overhead de parsing y unparsing.
- Problemas con tamaño de los buffers.

### 11.3.2 Call return

**Call return** es un estilo arquitectónico orientado a la programación estructurada. Su principal ventaja es su simplicidad y escasa longitud del código. Su desventaja es que su posibilidad de reuso es muy limitada.

En contraposición a este estilo, existen arquitecturas orientadas a objetos. Estas arquitectura implementan conceptos como

- *Information hiding*.
- Encapsulamiento.
- Herencia.
- Polimorfismo.

#### Ventajas:

- Reuso a gran escala.
- Todas las ventajas del encapsulamiento y el *information hiding*.
- Correspondencia en los objetos del dominio con objetos del mundo real.

#### Problemas:

- *Information hiding*: las decisiones de diseño que es probable que cambien son ocultadas en un módulo o un conjunto pequeño de módulo.

### 11.3.3 Layered

Las arquitecturas de estilo **layered**, se dividen en *layers* (o capas), cada una de las cuales:

- Oculta el nivel siguiente.
- Provee servicios al nivel anterior

En muchos casos el “bajar” de nivel implica acercarse al hardware o software de base. Los niveles van formando “virtual machines”.

#### Ventajas:

- Portabilidad.
- Facilidad de cambios.
- Reuso.

#### Problemas:

- Performace.
- La abstracción correcta puede ser difícil de encontrar.

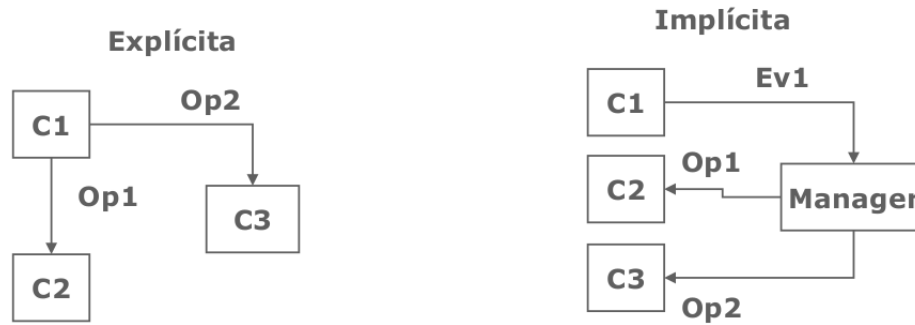
### 11.3.4 Client Server

En las arquitecturas que siguen el estilo **client server**, los componentes son clientes (que acceden a servicios) o servidores (que proveen servicios). Los servidores no conocen la cantidad o identidad de los clientes, mientras que un cliente si conoce la identidad del servidor a los que se conecta.

### 11.3.5 Componentes independientes

Las arquitecturas de **componentes componentes** son arquitecturas de procesos que se comunican a través del envío de mensajes. Los componentes son procesos independientes y los conectores representan envío de mensajes, que pueden ser sincrónicos o asincrónicos.

### Invocación implícita vs explícita



### 11.3.6 Centradas en datos

Las **arquitecturas centradas en datos** tienen una estructura de datos central (normalmente una base de datos) y componentes que acceden a ella. Gran parte de la comunicación está dada por esos datos compartidos.

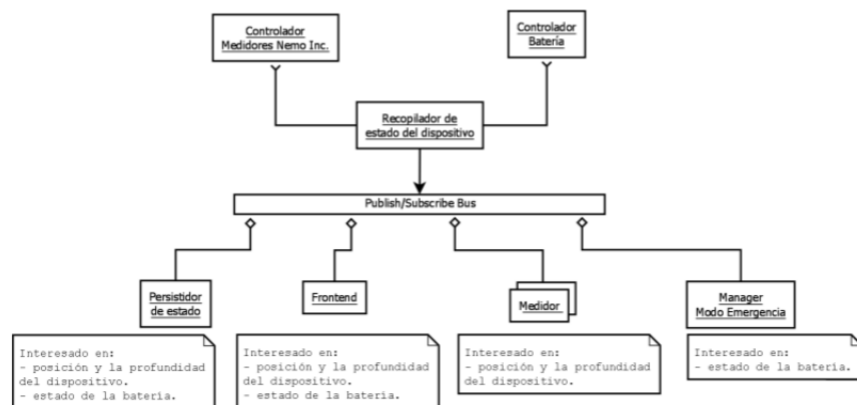
Las hay de dos tipos: *blackboard* y repositorio (cada vez más ocultos en un esquema tipo *layered*).

### 11.3.7 Basadas En Eventos

Las arquitecturas **basadas en eventos** se organizan como un grupo de componentes independientes con poco acoplamiento entre sí que permite comunicar a componentes a través de mensajes asíncronos.

Los componentes disparan la ejecución de otros componentes a través de conectores punto a punto (como *call asíncronico*) o conectores *multi-party*, en los que se envían mensajes a más de un destinatario. Un ejemplo de eso son los **publish-subscribe** en los que algunos componentes se suscriben por medio del conector a un conjunto de eventos proveniente de cualquier publicador y luego otros componentes envían eventos a un conjunto desconocido de receptores (ya suscriptos) a través del conector.

Ejemplo:



## 11.4 Viewtypes

Un sistema tiene varias estructuras, que pueden dividirse en tres grupos:

- De componentes y conectores: aquí los elementos son unidades de *run-time* independientes y los conectores son los mecanismos de comunicación entre esos componentes.

- De módulos: los módulos son unidades de implementación, una forma de ver al sistema basada en el código.
- Estructuras de asignación (“*allocation*”): Muestra la relación entre elementos de software y los elementos en uno o más entornos externos en los que el software se crea y ejecuta.

Llamamos **viewtypes** a las vistas de arquitecturas orientadas a estas tres estructuras.

#### 11.4.1 De módulos

Un **módulo** es una unidad de código que implementa un conjunto de responsabilidades (una clase, una colección de clases, una capa o cualquier descomposición de la unidad de código. Involucra nombres, responsabilidades, visibilidad de las interfaces, etc). La palabra “módulo” hace referencia a una unidad en tiempo de diseño.

Este viewtype involucra las siguientes relaciones:

- Descomposición (“es parte de”): los módulos tiene relación del tipo “es un submódulo de”. Representa la descomposición del código en sistemas, subsistemas, etc
- Usos (“depende de”): los módulos tienen relación del tipo “usa a”. Se dice que un módulo *A* usa a *B*, si la correcta ejecución de *B* es necesaria para la correcta ejecución de *A* (no es lo mismo que invocación). Esta vista hace explícito como una funcionalidad se mapea con una implementación, mostrando la relaciones existentes entre las distintas partes de código que finalmente la implementan.
- Clases (“es un”): los módulos en este caso son clases las relaciones son “hereda de” u otras asociaciones. Este estilo es útil cuando el arquitecto desea soportar la posible extensión y evolución de la arquitectura. Los módulos son organizados de modo de capturar concordancias y variaciones entre sí. El módulo padre reúne todas las concordancias entre sus hijos, y cada hijo posee las variaciones correspondientes.

Se usa UML para graficarlo.

#### Utilidad

- Construcción:
  - Puede proveer un blueprint para el código fuente.
  - Generalmente es posible establecer mapeos entre módulos y estructuras físicas (como archivos de código fuente).
- Análisis:
  - Trazabilidad de Requerimientos.
  - Analiza como los requerimientos funcionales son soportados por las responsabilidades de los distintos módulos.
- Análisis de Impacto:
  - Permite predecir el efecto de las modificación del sistema.
  - Comunicación.
  - Pueden ser utilizadas para explicar las funcionalidades del sistema a alguien no familiarizado con el mismo.

### Cuando no

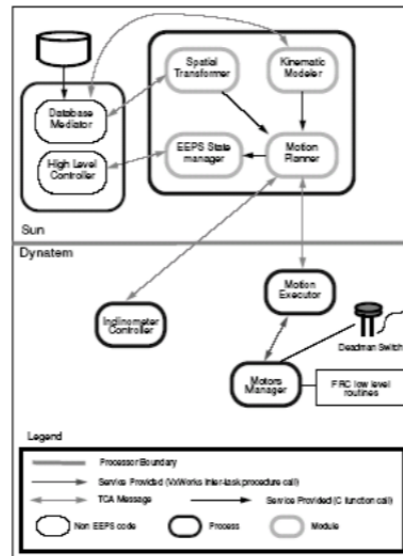
El viewtype de módulos no es de mucha utilidad para la realización de análisis de performance, confiabilidad u otras características asociadas al tiempo de ejecución.

Además, no se ven:

- Múltiples instancias de objetos.
- Relaciones existentes sólo en tiempo de ejecución.

### Ejemplo

Este es un ejemplo de viewtype combinado entre **módulos** y **componentes y conectores**.

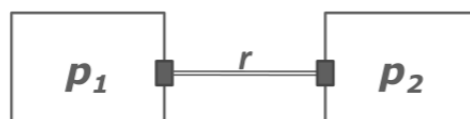


#### 11.4.2 De componentes y conectores

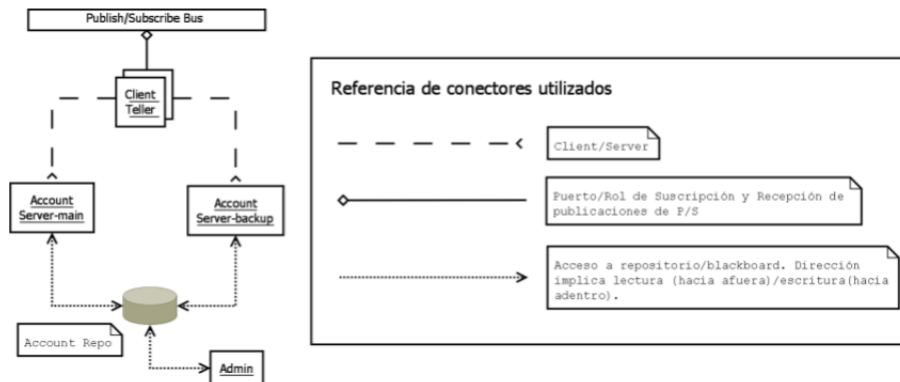
Las vistas de **componentes y conectores** están centradas en procesos que se comunican. Sus elementos son entidades con manifestación *runtime* que consumen recursos de ejecución y contribuyen al comportamiento en ejecución del sistema.

- La **configuración** del sistema es un grafo conformado por la asociación entre componentes y conectores.
- Las **entidades runtime** son instancias de tipos de conector o componente.
- Los **componentes** son entidades independientes y sólo se relacionan e interactúan a través de conectores. La palabra “conector” se refiere a unidades en tiempo de ejecución.
- Un **conector** es un componente que representa un camino en la interacción en tiempo de ejecución entre dos o más componentes. El tipo de conector indica la cardinalidad (cantidad de componentes en la interacción), las interfaces que soporta y las propiedades requeridas.

Formalmente siempre se asocian puertos de componentes con puertos de conectores (llamados roles): un puerto de componente  $p_1$ , es vinculado con un rol de conector  $r$ , si el componente interactúa sobre el conector usando la interfaz descrita por  $p_1$  y cumpliendo con la expectativas descritas por  $r$ .



## Ejemplo



### 11.4.3 De estructuras de asignación

La vista de **estructuras de asignación** involucra tres partes:

- **Deployment:** muestra cómo el software se asigna a hardware y elementos de comunicación.
- **Implementación:** muestra cómo los elementos de software se mapean a estructuras de archivos en repositorios de control de la configuración o entornos de desarrollo.
- **Asignación de trabajo :** asigna la responsabilidad del desarrollo y la implementación a equipos de programadores.

## 12 Tácticas

Una **táctica** busca controlar las respuestas a determinados estímulos. Representa una decisión de diseño. Informalmente, son las cosas que hago durante la etapa de diseño para lograr los atributos de calidad. Se define una **estrategia** como una colección de tácticas.

### 12.1 Relación con los estilos

Cada estilo puede implementar varias tácticas (Ej: el estilo *layer* implementa la táctica de *information hiding*). Cada una de las tácticas tiene un determinado impacto en los atributos de calidad (Ej: *information hiding* es una táctica para cubrir el atributo de modificabilidad).

### 12.2 Tácticas para los atributos

#### 12.2.1 Disponibilidad

- *Fault Detection:*
  - *Ping / Echo.*
  - *Heartbeat.*
  - *Exception.*
- *Recovery - Preparation and repair:*
  - *Voting.*

- *Active redundancy.*
  - *Passive redundancy.*
  - *Spare.*
- *Recovery - Reintroduction:*
  - *Shadow.*
  - *State resynchronization.*
  - *Rollback.*
- *Prevention:*
  - *Removal from service.*
  - *Transactions.*
  - *Process monitor.*

### 12.2.2 Performance

- *Resource Demand:*
  - *Increase computation efficiency.*
  - *Reduce computational overhead.*
  - *Manage event rate.*
  - *Control frequency of sampling.*
- *Resource Management:*
  - *Introduce concurrency.*
  - *Maintain multiple copies.*
  - *Increase available resources.*
- *Resource Arbitration:*
  - *Scheduling policy.*

### 12.2.3 Seguridad

- *Resisting Attacks:*
  - *Authenticate users.*
  - *Authorize users.*
  - *Maintain data confidentiality.*
  - *Maintain integrity.*
  - *Limit exposure.*
  - *Limit access.*
- *Detecting Attacks:*
  - *Intrusion detection.*
- *Recovering from an attack:*
  - *Restoration.*
  - *Identification: audit trail.*

#### 12.2.4 Modificabilidad

- *Localize Changes:*
  - *Semantic coherence.*
  - *Anticipate expected changes.*
  - *Generalize module.*
  - *Limit possible options.*
  - *Abstract common services.*
- *Prevention of Ripple Effects:*
  - *Hide information.*
  - *Maintain existing interface.*
  - *Restrict communication paths.*
  - *User an intermediary.*
- *Defer Binding Times:*
  - *Runtime registration.*
  - *Configuration files.*
  - *Polymorphism.*
  - *Component replacement.*
  - *Adherence to defined protocol.*

#### 12.2.5 Usabilidad

- *Sparate user interface.*
- *Support user Initiative:*
  - *Cancel.*
  - *Undo.*
  - *Aggregate.*
- *Support System Initiative:*
  - *User model.*
  - *System model.*
  - *Task model.*

#### Reglas de oro de la usabilidad

- Diálogos simples y naturales.
- Hablar el lenguaje del usuario.
- Minimizar la carga de memoria.
- Ser consistente.
- Proveer feedback.

- Proveer salidas marcadas claramente.
- Proveer atajos.
- Dar buenos mensajes de error.
- Prevenir y manejar errores.
- Ayuda y documentación.

## 13 Fuentes

- Slides de la cátedra de Ingeniería del Software II. Santiago Ceria.
- Apuntes de clase de Julián Sackmann.
- Archivos de audio de clases grabadas.